N84-31955

NASA Contractor Report 165874

Fault Tolerant Software Modules for SIFT

Myron Hecht and Herbert Hecht
SoHaR, Inc.
Los Angeles, California 90035

Contract NAS1-15428
July 1982

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

NASA Contractor Report 165874

Fault Tolerant Software Modules for SIFT

Myron Hecht and Herbert Hecht
SoHaR, Inc.
Los Angeles, California 90035

Contract NAS1-15428
July 1982

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION 1 - INTRODUCTION

This is the Final Engineering Report prepared for SRI International under Subcontract No. 14395 covering the implementation of software fault tolerance for critical modules of the SIFT operating software. The SIFT (Software Implemented Fault Tolerance) is an advanced computer concept developed by SRI for the NASA Langley Research Center under Contract NAS1-15428 to support the computational and reliability requirements of advanced fly-by-wire transport aircraft.

This report complies with the requirements of Article IV Item D of SRI Subcontract No. 14395.

Although this project constituted only a minor part of the SIFT effort, considerable advances in concepts and implementation of software fault tolerance were achieve under it. These are summarized in the paragraphs immediately following. Part 1.2 of the Introduction provides an overview of the specific modules for which fault tolerant designs were generated, the error reporter and the global executive. Part 1.3 describes the organization of the body of this report, and Part 1.4 acknowledges the contribution of individuals outside our organization to this work.

1.1 ADVANCES IN SOFTWARE FAULT TOLERANCE IN THIS EFFORT

Because the software in the SIFT operating system is essential for both scheduling of application tasks and recovery from hardware failures, special efforts have been made to verify this software in a formal manner. In addition, it is being subjected to an extensive test program. Nevertheless, provision of fault tolerance features was deemed desirable for selected portions of these programs that have a key role in the recovery from failures. Note that this software must perform in accordance with its specification in the presence of faults in one or more of the component computers of SIFT or in their interconnections.

The fault tolerance technique selected for this purpose is that of the recovery block [RAND75]. Specific implementations of this technique to real-time applications and the transport aircraft environment had already been described prior to the effort reported here [HECH76, AERO78]. The basic structure for a recovery block is

```
Ensure T
        By P
        Else by Q
Else Error
```

where T is an acceptance test condition, i. e. a condition which is expected to be met by succesful execution of either the primary routine P or the alternate Q. The internal control of the recovery block transfers to Q if the test condition is not met by executing P.

The effectiveness of the fault tolerance provisons depends on the coverage of the acceptance test and the avoidance of correlated failure mechanisms in P and Q. Prior work had dealt primarily with software associated with a physical process (e. g., attitude control), where the environment could be depended on to furnish clues on the 'true' state of the process (e. g., by means of sensors independent of those that furnished the primary input data).

The uses served by the fault tolerant modules for SIFT are of an intrinsically logical nature, dealing with the reporting of errors and the action to be taken after positive reports. For applications of this type, the environment does not furnish independent clues, and the 'truth' has to be teased out of the logical process itself. Athough the routines to which fault tolerance was applied were quite small, the work was therefore quite challenging. The main contribution of the effort reported here to the field of fault tolerant software is the evolution of a technique for formulating acceptance tests in logic oriented applications based on conditions that are inherently orthogonal to the logic implemented by the primary routine. A very clear example of this technique is presented in the acceptance test for the error reporter in 2.2.

Further contributions will be found in the use of fault trees to identify the requirements for acceptance tests and to determine the completeness of the coverage of these tests. Some limitations of the recovery block technique were encountered in constructing alternate routines that are truly independent of the primary ones (and also of the acceptance test) for applications in which the principal operations are addition and subtraction (comparison). In all cases it was at least possible to change the order of operations and thereby to avoid common sequence dependent failures. Greater independence might be achievable by permitting alternate routines a larger scope (i. e., by letting one alternate routine perform the computations carried out in several primary routines). This concept seems worthy of exploration in future studies.

1.2. OVERVIEW OF THE FAULT TOLERANT ERROR REPORTER AND GLOBAL EXECUTIVE

SIFT achieves its high reliability by use of multiple processors with an excess of computing capacity. When a single processor fails, it is configured out of the system, a measure which ensures survival of the computer as a whole. Thus, an important function of the SIFT operating system is the retiring of faulty processors. A processor is defined as faulty if its output differs from those of other processors for a given task. The SIFT error reporter and global executive tasks collect information on disagreeing processors, process it, and designate processors for retirement to the reconfiguration task.

The error reporter analyzes error data collected by the voter to determine what processors appear to be faulty and indicates these in an error report. Because a processor can not report itself as faulty (even if the voter data would tend to indict it), error reports from each processor may differ. The global executive reviews all error reports, and if two or more processors point to a third as being faulty, then the result is transmitted to the reconfiguration task.

The error reporter and global executive have been made fault tolerant by applying the recovery block principle described in section 1.1. Both tasks have an acceptance test and recovery block associated with them. Thus, there now exists a primary error reporter and global executive as well as alternates. Very few changes were necessary to the primary routines in order to implement the recovery blocks, and, with the exception of the addition of a single integer variable, no changes were made to the remainder of the system software.

As noted above, the error reporter acceptance test establishes that all processors with an excessive number of disagreements with the voter output are detected, and ensures that no properly functioning processors are designated as faulty. The alternate error reporter operates independently of the primary routine, but produces an identical output. The acceptance test involves approximately twenty PASCAL statements, and the length of the alternate error reporter is approximately the same as the primary. Thus, neither routine will have a significant effect on the timing of the SIFT operating system.

The global executive acceptance test is coded in two modules: the first, which is run before the primary routine, verifies that all input to the global executive is current, and the second, which is run after the primary global executive, checks for correct execution. If errors are detected by either module of the acceptance test, the alternate global executive is invoked.

Execution of each of these routines is checked by the other. Thus, the global executive checks on the execution of the error reporter acceptance test on each processor by means of the frame count encoded in the error words. Similarly, an output of the global executive which also has a frame count encoded within it is checked by the error reporter in the subsequent frame. Notification to the system is provided in the case of either error.

In addition to verifying correct execution of their immediately associated primary routines, these acceptance tests can be expanded to give some indication of the functioning of the reconfiguration task. If a processor indicated as not working in the system status vector is generating error reports, then obviously, it has not retired. Although diagnosis of the discrepancy is beyond the scope of the tasks of the software developed here, an indication is made to the system that an off-normal condition exists, and appropriate action can be taken by the operating system.

A major portion of the coding effort went toward the validation of the five Pascal procedures developed as part of the error reporter and global executive recovery blocks. Driver routines with approximately 8 to 10 times the amount of code in these routines were developed in order to adequately support the large number of test cases which had to be run during validation.

1.3. ORGANIZATION OF THE REPORT

Section 2 describes the fault tolerant error reporter. Included are a description of the acceptance test, the error conditions which it covers, a description of the alternate routine, implementation requirements for integration of the fault tolerant error reporter into the operating system, and a description of the software validation. Section 3, which describes the fault tolerant global executive, has a similar organization.

1.4 ACKNOWLEDGEMENTS

The authors wish to express their appreciation for the cooperation received in this effort from personnel of SRI International and of the NASA Langley Research Center. Mr. Jack Goldberg gave guidance and support throughout this work and was particularly helpful in pointing out from time to time that there was a forest when the effort seemed directed at individual trees, twigs or even smaller manifestations of nature's bounty. Drs. Charles Weinstock and P. Michael Mellar-Smith helped with information on the primary SIFT software, on the environment in which this operated, and on the interfaces which had to be observed in the design of the fault tolerance provisions. To Mr. Billy L. Dove and Mr. Nicholas D. Murray our thanks for the support of this work and for permitting us to participate in an important area of fault tolerant computing.

SECTION 2: ERROR REPORTER

The voter routine of each processor in SIFT maintains its own record of the number of disagreements from the majority of all other processors. The SIFT error reporter marks processors as being faulty based on the disagreement count generated by the voter. The error reporter acceptance test compares the number of recorded processor disagreements with the output of the error reporter, and if processors are incorrectly characterized as working or failed, it invokes the alternate routine.

2.1. ERROR REPORTER ACCEPTANCE TEST

The SIFT voter routine marks individual processor disagreements from the majority in an array designated as errors. The error reporter sets a bit in a word called err for each processor with an excessive number of disagreements as reported in errors. Bits 0 through 7 in err represent the correspondingly numbered processors. The acceptance test checks that te error reporter was invoked in the previous subframe, and calls the alternate error reporter upon detection of a discrepancy between err and errors.

Figure 2.1 is a flow chart of the proposed error reporter acceptance test, and figure 2.2 is a Pascal listing of the procedure which has been developed and tested. The test counts the number of non-disagreeing processors in a counter designated as right and outvoted processors in a counter designated as wrong. It then checks the number of disagreements and the operational status of every processor designated as faulty. A Boolean variable to invoke the alternate error reporter is set to TRUE if a working processor marked as faulty has fewer than the threshold number of disagreements. The final segment adds right and wrong; if this sum does not equal the total number of processors, the acceptance test will invoke the alternate error reporter.

If the error reporter acceptance test does not detect any failures, it writes the frame count in the 8 most significant bits of err. When the global executive acceptance test checks these bits for the frame count, it will verify that the error reporter acceptance test has been executed in the current frame, and that consequently, err reports are current. If a discrepancy between the current frame and that encoded in the 8 most significant bits of err from a particular processor is encountered, the global executive sets a corresponding bit in an integer variable called mismatch along with the frame count in the 8 most significant bits. The error reporter acceptance test will then increment errors in the appropriate position in the subsequent frame. Thus, failure to execute the error reporter in the current frame will increase the likelihood that the processor will be retired by the alternate error reporter.

2.2. COVERAGE OF THE ERROR REPORTER ACCEPTANCE TEST

The error reporter acceptance test detects the following faults:

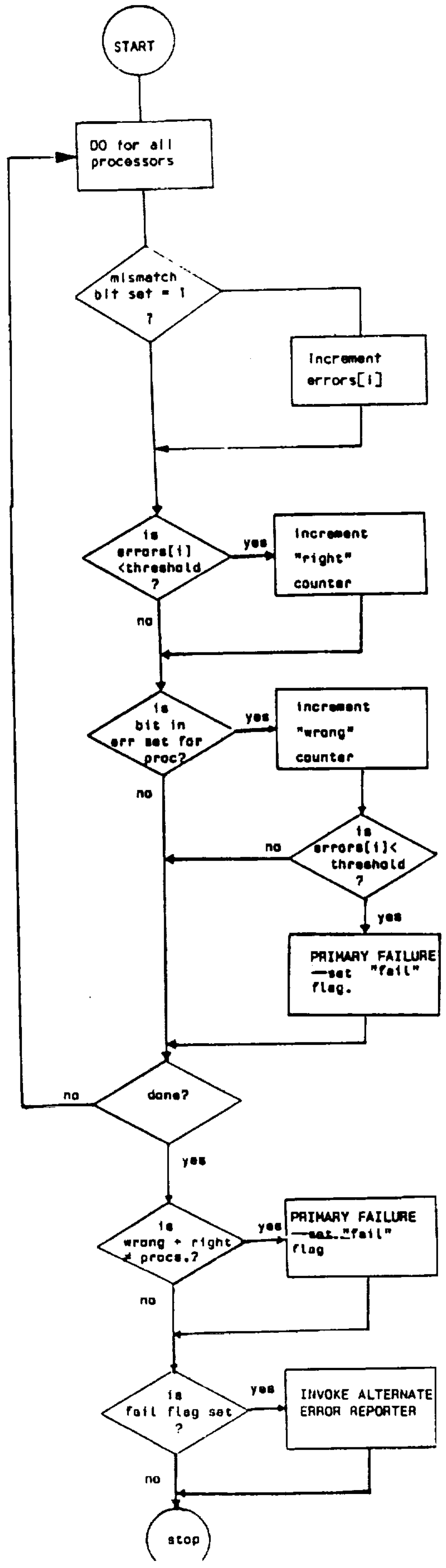
START
DO for all processors
mismatch bit set = 1 ?
increment errors[i]
is errors[i] <threshold ?
yes
no
increment "right" counter
is bit in err set for proc?
yes
no
increment "wrong" counter
is errors[i]< threshold ?
no
yes
PRIMARY FAILURE —set "fail" flag.
done?
no
yes
is wrong + right ≠ procs.?
yes
no
PRIMARY FAILURE —set "fail" flag
is fail flag set ?
yes
no
INVOKE ALTERNATE ERROR REPORTER
stop

Figure 2.1. Flow chart of Error Reporter Acceptance Test

```
PROCEDURE ACCEPTANCE_TEST;
(*error reporter acceptance test*)
VAR
        EXCOUNT,WRONG,RIGHT,DIVISOR,CHECK,I,J,MISM:INTEGER;
        FAILFLG:BOOLEAN;

begin

        excount:= mismatch div 256;
                          (*check execution count of global exec*)
        if (framecount mod 256)<>(excount - 1) then erfails:=true;
                         (*erfails is a global variable which notifies
                           the system that the global exec has not run*)
        mism:=mismatch div 256;
        wrong:=0;
        failflg:=false;
        right:=0;
        divisor:=1;
        for j:=0 to maxprocessors do   (*check for omission errors*)
                begin
                mism:=mism div divisor
                        (*processor has 1 strike against it if
                          error reporter didn't run in prev. frame*)
                if odd (mism) then errors[j] :=errors[j] +1;
                if (errors[j]<threshold) and (working[j])
                 then right:=right+1;
                                        (*count for omissions test*)
                check:=err div divisor;
                                        (*shift err appropriate
                                          no. of places to the right*)
                if odd(check) then begin
                        wrong:=wrong+1;  (*count for omissions test*)
                        if (errors[j]<threshold) and (working[j])
                        then failflg:=true (*check for false positives*)
                end;
                divisor:=divisor*2;
        end;
        if wrong+right<>maxprocessors +1 then failflg:=true;
                                (*omissions test*)
        if failflg then alt_error_reporter
        else err:=err + 256 * (framecount mod 256);
end;
*
```

Figure 2.2. Pascal listing of Error Reporter Acceptance Test

(1) failure to invoke the error reporter during each frame

(2) failure to report processors with an excessive number of disagreements as faulty to the global executive, and

(3) designation of a properly functioning processor as faulty

The validity of the input to the test (e.g. framecount, working, and errors) is not checked, and it is possible that errors in these variables could be propagated into err. However, to a certain extent, these failures are covered by other processor error reports in the global executive.

The primary consideration in the design of this acceptance test was that the verification and failure detection be performed in a manner independent of the primary error reporter. the following subsections describe the means by which the errors listed above are detected.

2.2.1. Failure to Execute During Each Frame

As noted above, the global executive acceptance test checks the frame count mod 256 encoded in the front part of each error report. Consequences of the failure to execute the error reporter on a given processor are limited; a consistent pattern of failures will be detected by means of the error reports of other processors. Discrepancies will ultimately lead to the retiring of processors which do not execute the error reporter. The present acceptance test implementation calls for the retirement of the processor if any other discrepancy from the system (i.e. voter) output occurs.

Just as the global executive checks execution of the error reporter, the converse also occurs. If the frame count encoded in the front eight bits of mismatch minus the frame count mod 256 is not equal to 1, then the global executive acceptance test has not been executed in the previous frame, and the system is notifed. Failure to execute the global executive may result in more serious consequences than failure to execute the error reporter, and the "one count against you" strategy described in the previous paragraph is not appropriate.

2.2.2. Failure to Report Processors with an Excessive Number of Disagreements to the Global Executive.

In order to achieve independence from the primary error reporter algorithm, the acceptance test checks for this failure indirectly by testing for the following conditions:

(a) the total number of processors reported as faulty is correct, and

(b) all processors designated as faulty have greater than the threshold number of disagreements

In this acceptance test, the number of processors with less than the threshold number of disagreements is counted in a variable designated as right, and the number having excess disagreements are counted on a second counter labeld wrong. If the sum of wrong and right is equal to the total number of processors, then the error reporter can be shown to have performed correctly when the third part of the acceptance test, described in the following section, has not detected any failures. This acceptance test is a particularly clear example of using algorithms which are orthogonal to the primary routine.

2.2.3. Designated a Properly Functioning Processor as Faulty

The final part of the acceptance test is to ensure that all processors designated as malfunctioning have at least the threshold number of disagreements. This determination is made by checking the number of disagreements of these processors. If any values of the array are below the threshold for working processors marked as faulty, then the primary error reporter has failed, and the alternate is invoked.

2.3. ALTERNATE ERROR REPORTER

Independence in the structure and operation from the primary error reporter was a chief objective in the alternate routine design. In addition, its output had to be compatible with the global executive.

These requirements resulted in a routine which is essentially the inverse of the primary error reporter. An alternate error word, designated as erra, is initially set ot all 1's; the alternate error reporter sets erra bits to 0 if the number of disagreements in the appropriate element of the errors array is less than the threshold. If there are more bits in erra than there are processors (e.g. if there are six processors and eight bits in erra), the leading bits are set to 0. Finally, the primary error word, err, is set equal to erra, loaded with frame count information, and placed in the pre-broadcast buffer. The complementary nature of this routine is maintained in the order of setting the error word bits -- the processors are checked in ascending order rather than the descending order used in the primary error report.Figure 2.4.is a Pascal listing of the alternate error report.

2.4. IMPLEMENTATION REQUIREMENTS

As noted previously, the acceptance test and the alternate error reporter are short and relatively simple procedures which were written to be compatible with the SIFT operating system. Additional local variables are required as shown in the listings for the error reporter acceptance test and alternate routine. In addition, some modifications to the primary error reporter are necessary to enable it to transmit processor states to the global executive and execution information to the acceptance test. No changes in the broadcasting protocol are required.

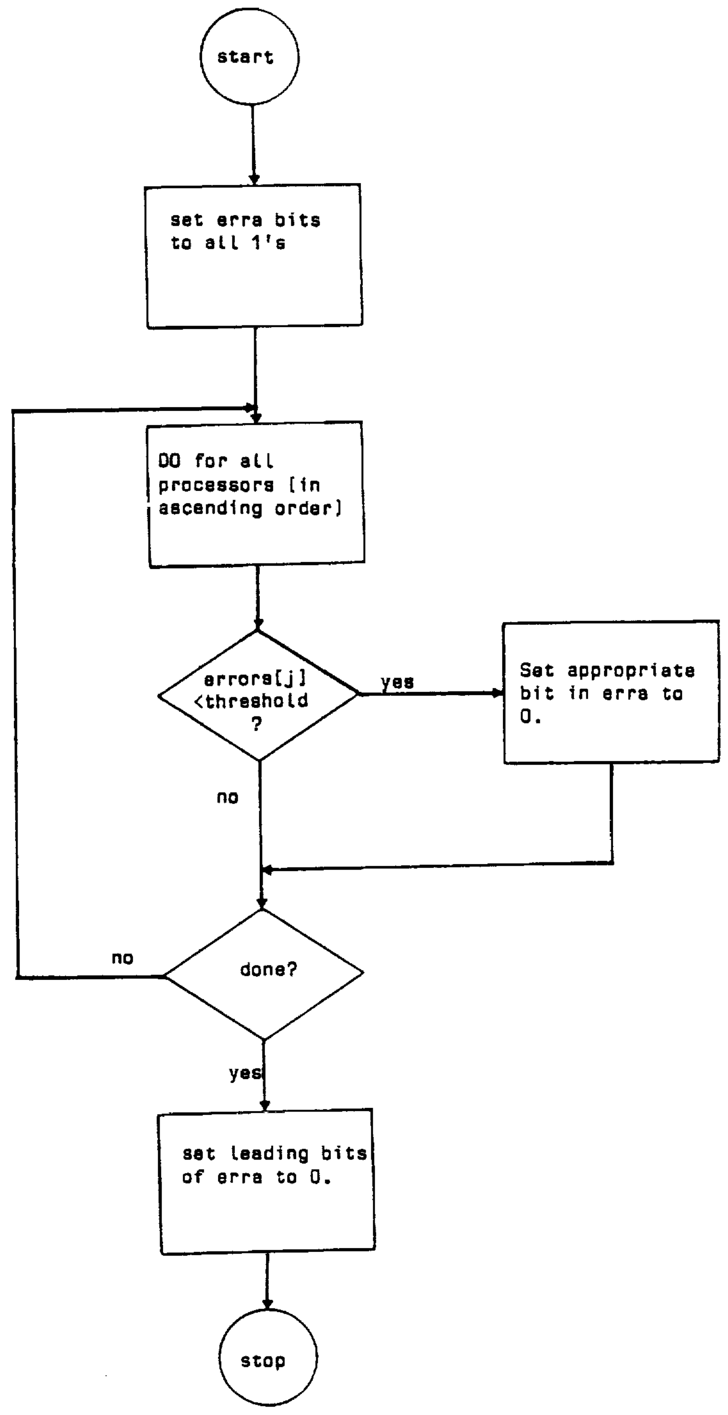
start
set erra bits to all 1's
DO for all processors [in ascending order]
errors[j] <threshold ?
yes
Set appropriate bit in erra to 0.
no
no
done?
yes
set leading bits of erra to 0.
stop

Figure 2.3. Flow chart of Alternate Error Reporter

```
PROCEDURE ALT_ERROR_REPORTER;
(*this is the alterate error reporter*)

CONST
        ALLONES=377B;
VAR
        ERRA:INTEGER;  (*alternate error word*)
        I,K:INTEGER;

begin
        erra:=allones;
        k:=1;
        for i:=0 to maxprocessors do
                begin
                 if (errors[i]<threshold) and (working[i])
                  then erra:=erra-k;
                 k:=k*2;
        end;
        erra:=erra - (allones - k + 1);    (*remove leading bits*)
        err:=erra + 256*sfcount;
        prebroadcast(errerr,err);
end;
*
```

Figure 2.4. Pascal listing of alternate error reporter

2.5. ERROR REPORTER RECOVERY BLOCK VALIDATION

The major objective of the testing performed on the error reporter recovery block was to provide a comprehensive set of cases which would demonstrate satisfactory performance when the error reporter was functioning properly and when it had failed. Figure 2.5shows the top level fault tree that was used to define this set. The recovery block fails if the primary error reporter fails without detection by the acceptance test, or if the alternate fails after being invoked by the error reporter acceptance test. Failure due to an undetected primary routine fault will occur when both the primary routine fails and the acceptance test does not detect it. The same potential failures affect the acceptance test and the alternate routine and thus, they were both validated simulataneously.

Figure 2.6 continues the development. There are two major classes of errors: failure to identify a processor with excess disagreements, and reporting a processor with less than two disagreements in the error report. Under the first class of errors, one, two, or three processors could remain unidentified. Further expansion of the tree shows that failure to identify two outvoted processors is caused by failure to identify the first process _and_ failure to identify the second. Similarly, failure to identify three processors having excess disagreements can be broken down into failure to identify the first processor _and_ failure to identify the second _and_ failure to identify the third.

Figure 2.7 continues this development. Any of the six processors could be identified as the first failure. Once the validation has established that the error reporter acceptance test and alternate can correctly identify the first error committed by the primary routine (i.e. failure to identify one processor with an excess number of disagreements), validation for the condition of two outvoted processors can be performed by holding fixed the first processor with excess disagreeements and only varying the second. Thus, processor 0 is assigned the first error, and processors 1 through five are each, in turn, given an excess number of disagreements in the _errors_ array. Similar logic applies to the third and fourth processors with excess disagreements.

Figure 2.8 is a further development of the fault tree which summarizes the pattern in which the processors are tested. Transfer 1011 shows that all six processors are tested for the case in which the primary error reporter fails to detect one processor with excess disagreements. Transfer 1012 shows that when two disagree excessively, the primary error reporter is always assumed to have dectected an excess disagreement condition in processor 0, and that the acceptance test and alternate are tested with the second error in processors 1 through 5. For failure of the primary error reporter to detect a third excessively disagreeing processor, transfer 1013 shows that processors 0 and 1 are assumed to be the first two, and the third occurs in processor 2, 3, 4, or 5. Finally, for four errors, processors 0, 1, and 2 are assumed to have excess disagreements, and the final error varies between processors 3, 4, and 5.

The fault tree for the second class of errors, spurious identification of correctly functionining processors as having excessive disagreements is shown in figures 2.8 and 2.9. Incorrect identification of a processor as malfunctioning can occur when there are either no disagreements or a single disagreement.

Incorrect characterization of the processor can also occur when there are one, two, or three other processors which actually have excessively disagreed with the voter output. As previously, not all processors need to be considered. The testing scheme in this case is to ensure that the error reporter acceptance test can detect a false failure of each processor when any other processor has failed. Table 2.1 is a list of the validation tests required to verify the correctness of the error reporter acceptance test and alternate executive based on the fault trees described here.

Complete test required simulation of a major portion of the SIFT operating system. The simulation program, called DRIVER, prepares the *errors* and *working* arrays of the voter and *err* word of the error reporter based on external inputs. It next invokes the acceptance test, outputs results, and invokes the alternate if an error is detected. Appendix A shows a complete listing of the program.

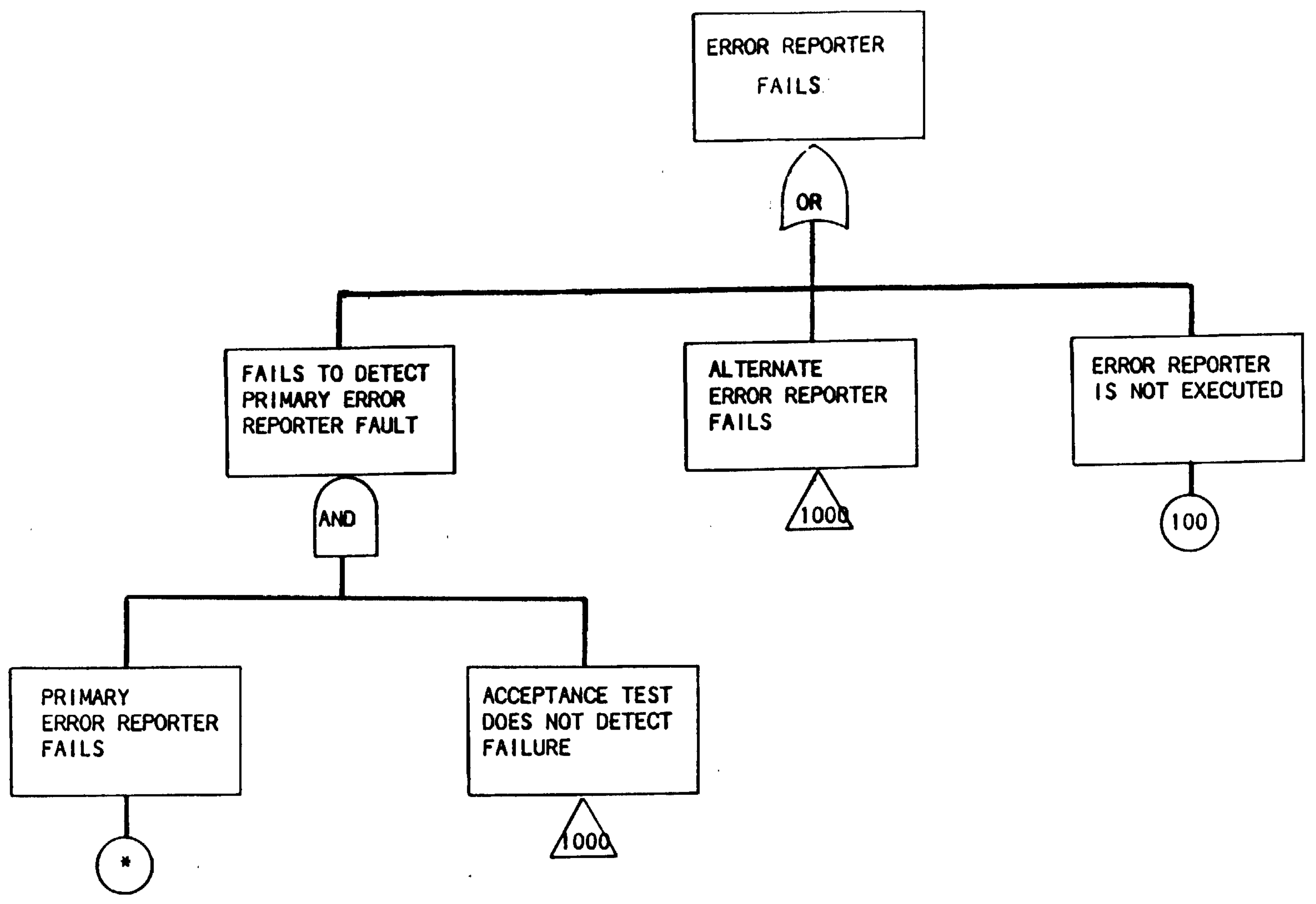
ERROR REPORTER FAILS
OR
FAILS TO DETECT PRIMARY ERROR REPORTER FAULT
ALTERNATE ERROR REPORTER FAILS
ERROR REPORTER IS NOT EXECUTED
AND
1000
100
PRIMARY ERROR REPORTER FAILS
ACCEPTANCE TEST DOES NOT DETECT FAILURE
*
1000

Figure 2.5. Top level tree for Error Reporter Failures

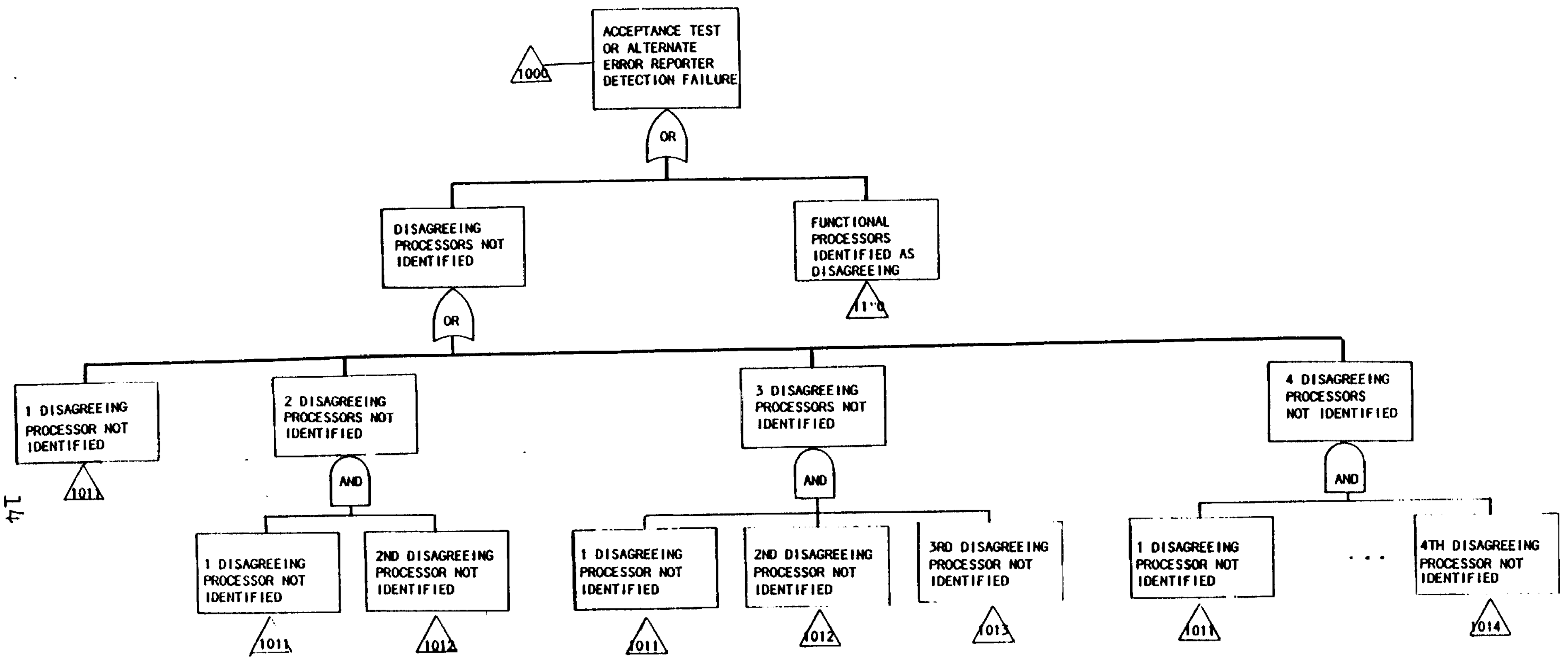
ACCEPTANCE TEST OR ALTERNATE ERROR REPORTER DETECTION FAILURE
1000
OR
DISAGREEING PROCESSORS NOT IDENTIFIED
FUNCTIONAL PROCESSORS IDENTIFIED AS DISAGREEING
11"0
OR
1 DISAGREEING PROCESSOR NOT IDENTIFIED
1011
2 DISAGREEING PROCESSORS NOT IDENTIFIED
AND
1 DISAGREEING PROCESSOR NOT IDENTIFIED
1011
2ND DISAGREEING PROCESSOR NOT IDENTIFIED
1012
3 DISAGREEING PROCESSORS NOT IDENTIFIED
AND
1 DISAGREEING PROCESSOR NOT IDENTIFIED
1011
2ND DISAGREEING PROCESSOR NOT IDENTIFIED
1012
3RD DISAGREEING PROCESSOR NOT IDENTIFIED
1013
4 DISAGREEING PROCESSORS NOT IDENTIFIED
AND
1 DISAGREEING PROCESSOR NOT IDENTIFIED
1011
. . .
4TH DISAGREEING PROCESSOR NOT IDENTIFIED
1014

Figure 2.6. Classes of Error Reporter Failures

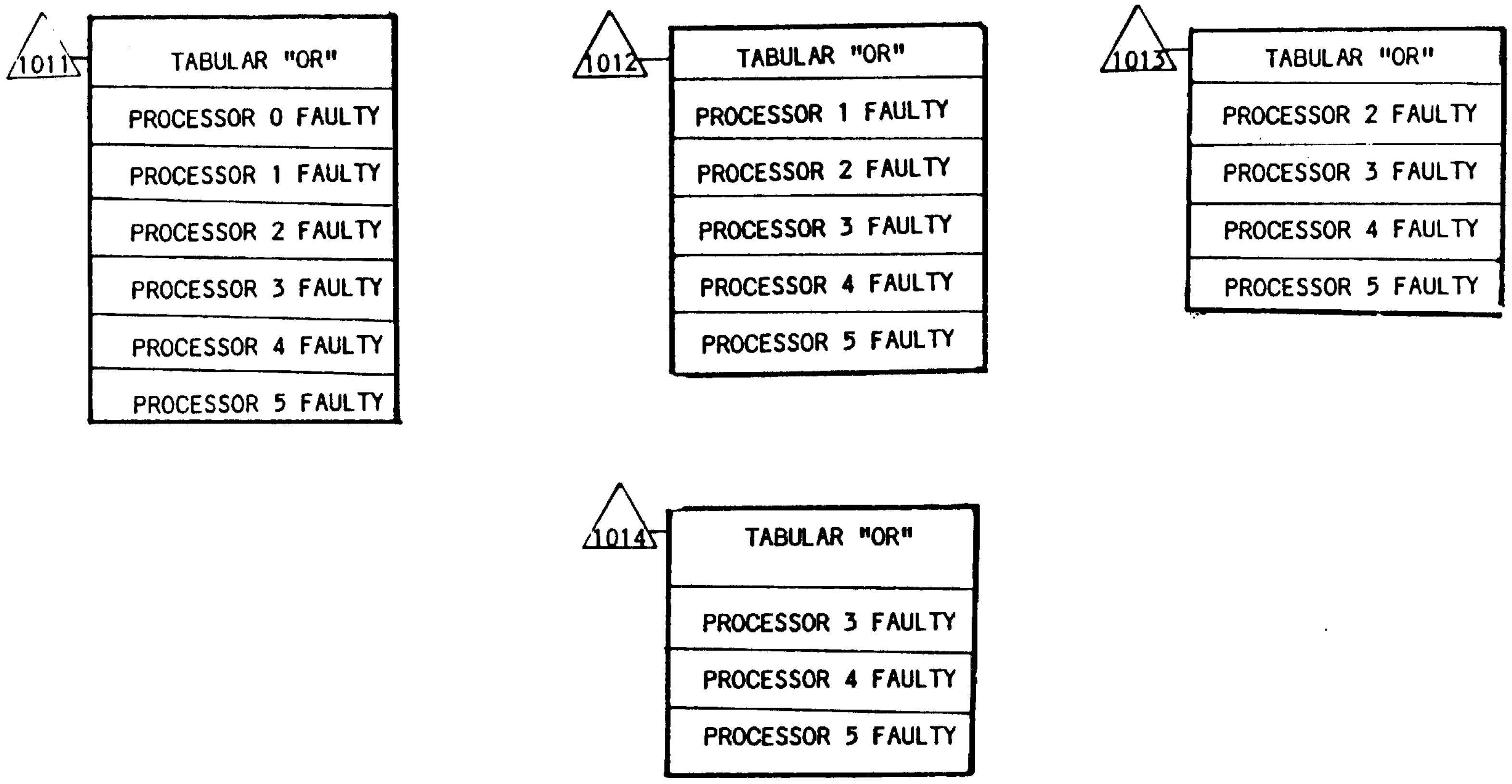

Figure 2.7. SIFT configurations used for detection failure validations

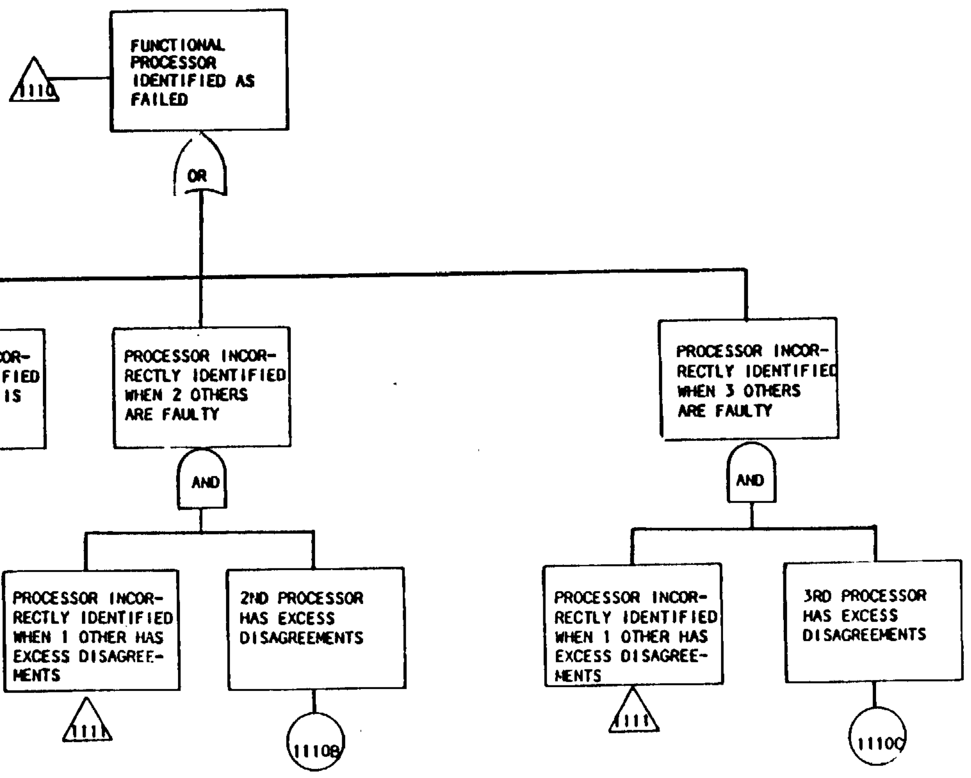
FUNCTIONAL PROCESSOR IDENTIFIED AS FAILED
1110
OR
PROCESSOR INCORRECTLY IDENTIFIED WHEN NO OTHERS ARE FAULTY
1110A
PROCESSOR INCORRECTLY IDENTIFIED WHEN 1 OTHER IS FAULTY
1111
PROCESSOR INCORRECTLY IDENTIFIED WHEN 2 OTHERS ARE FAULTY
AND
PROCESSOR INCORRECTLY IDENTIFIED WHEN 1 OTHER HAS EXCESS DISAGREEMENTS
1111
2ND PROCESSOR HAS EXCESS DISAGREEMENTS
1110B
PROCESSOR INCORRECTLY IDENTIFIED WHEN 3 OTHERS ARE FAULTY
AND
PROCESSOR INCORRECTLY IDENTIFIED WHEN 1 OTHER HAS EXCESS DISAGREEMENTS
1111
3RD PROCESSOR HAS EXCESS DISAGREEMENTS
1110C

Figure 2.8. Incorrect Characterization of a Functional Processor as Failed

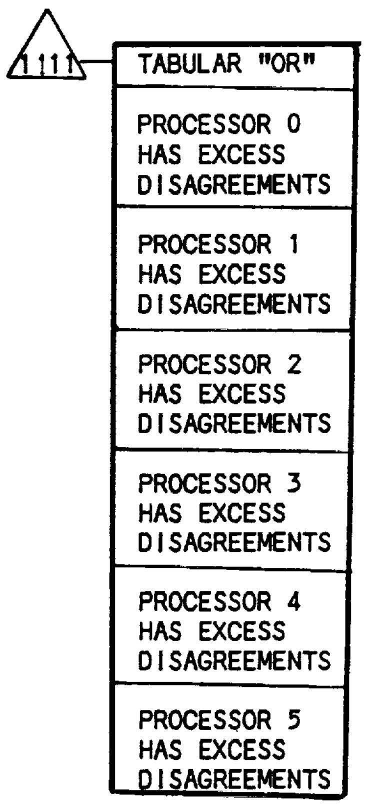
1111
TABULAR "OR"
PROCESSOR 0 HAS EXCESS DISAGREEMENTS
PROCESSOR 1 HAS EXCESS DISAGREEMENTS
PROCESSOR 2 HAS EXCESS DISAGREEMENTS
PROCESSOR 3 HAS EXCESS DISAGREEMENTS
PROCESSOR 4 HAS EXCESS DISAGREEMENTS
PROCESSOR 5 HAS EXCESS DISAGREEMENTS

Figure 2.9. Final Development of Figure 2.8

TABLE 2.1. FAULTS FOR WHICH VALIDATION TESTING IS REQUIRED FOR THE ERROR REPORTER ACCEPTANCE TEST AND ALTERNATE ERROR REPORTER

Fault Tree Designation	Description
1011	Failure to detect primary error reporter's not identifying a single processor as having excess disagreements
1012	Failure to detect primary error reporter's not identifying a second processor as having excess disagreements given that the first has been identified
1013	Failure to detect primary error reporter's not identifying a third processor as having excess disagreements given that the first two have been identified
1014	Failure to detect primary error reporter's not identifying a fourth processor as having excess disagreements given that the first two have been identified.
1110A	Failure to detect primary error reporter's false identification of a functional processor as having excess disagreements given that no other processor has failed.
1110B	As 1110A, given that 1 other processor failed.
1110C	As 1110A, given that 2 other processors failed.
1110D	As 1110A, given that 3 other processors failed.
1111	Failure to detect primary error reporter's false identification of a functional processor as having excess disagreements given that one other processor has failed.

SECTION 3: GLOBAL EXECUTIVE

This section describes the acceptance test and alternate routine for the SIFT global executive. The acceptance test is coded in two modules: the first, which is run before the primary routine, verifies that all input to the global executive is current, and the second, which is run after the primary global executive, checks for correct execution. If execution errors are detected by either module of the acceptance test, the alternate global executive is invoked.

3.1. GLOBAL EXECUTIVE ACCEPTANCE TEST

The August, 1980 version of the SIFT operating system has the error reports for the active processors contained in the array prevote[errerr,*] where errerr is a constant set to 1. The error reports themselves are contained within the 8 least significant bits of each 16-bit element of prevote, and the frame count is encoded in the 8 most significant bits by means of the error reporter acceptance test. The global executive reads successive bits of each prevote element by shifting the word to the right. Because of this destructive read, it is necessary to reproduce the error report information prior to execution of the primary routine. This task is performed by the first module of the global executive acceptance test designated PREGEXEC. PREGEXEC also checks on the frame count which has been encoded by the error reporter acceptance test. After execution of the primary global executive, the second module of the global executive acceptance test, called GEXECTEST, is executed. GEXECTEST checks each position of each word in an order orthogonal to the primary global executive. It then compares this result with the appropriate bit in RECONF, the retirement word generated by the primary routine. If there is a discrepancy, the alternate global executive, ALTGEXEC, is called. ALTGEXEC is described in section 3.

Figures 3.1 and 3.2 are flow charts of the two modules of the global executive acceptance tests,and figure 3.3 contains the corresponding listings. The first module of the acceptance test, PREGEXEC, checks the framecount contained in the most significant 8 bits of each error report which have been written by the error reporter acceptance test, and then recopies the least significant half of the word into the most significant position in order to preserve them for the second module of the global executive acceptance test.

Those error words containing frame counts different from that of the system are set to zero as a means of masking them from the global executive, and a failure counter for the processor error report is incremented. The subsequent execution of the error reporter on other processors will count this indicator as a disagreement when writing their reports, an action which will result in retirement of this processor if at least one other discrepancy is detected.

Once the primary global executive has been run, the second module of the acceptance test checks the correctness of its execution, and invokes the alternate routine upon detection of an error. A major consideration in the design of the acceptance test was that it be independent of the primary routine. Thus, whereas the primary checks each position of an error report before moving on to the next, the acceptance test checks a given position of all error reports

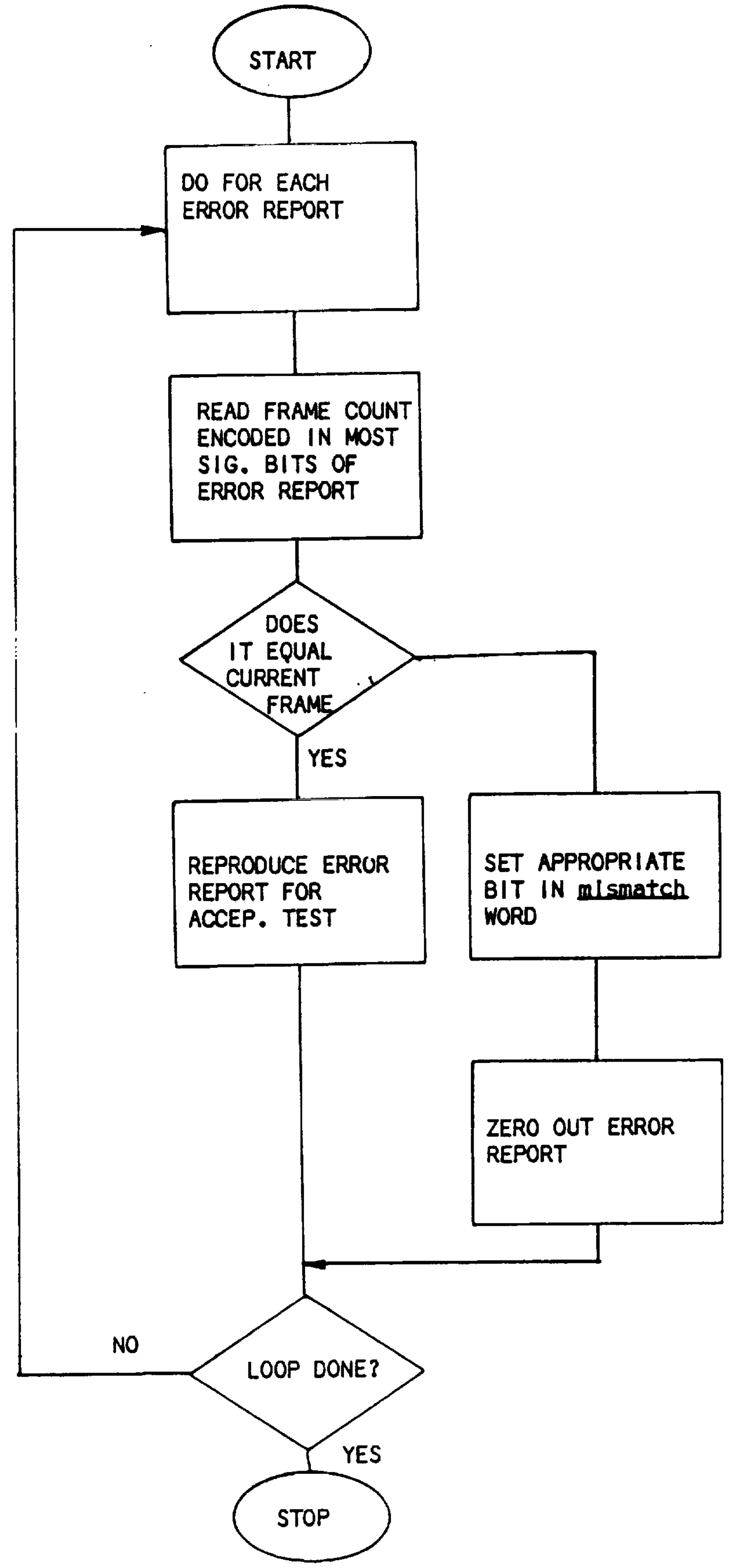
START
DO FOR EACH ERROR REPORT
READ FRAME COUNT ENCODED IN MOST SIG. BITS OF ERROR REPORT
DOES IT EQUAL CURRENT FRAME
YES
REPRODUCE ERROR REPORT FOR ACCEP. TEST
SET APPROPRIATE BIT IN mismatch WORD
ZERO OUT ERROR REPORT
NO
LOOP DONE?
YES
STOP

Figure 3.1. Flow chart for PREGEXEC

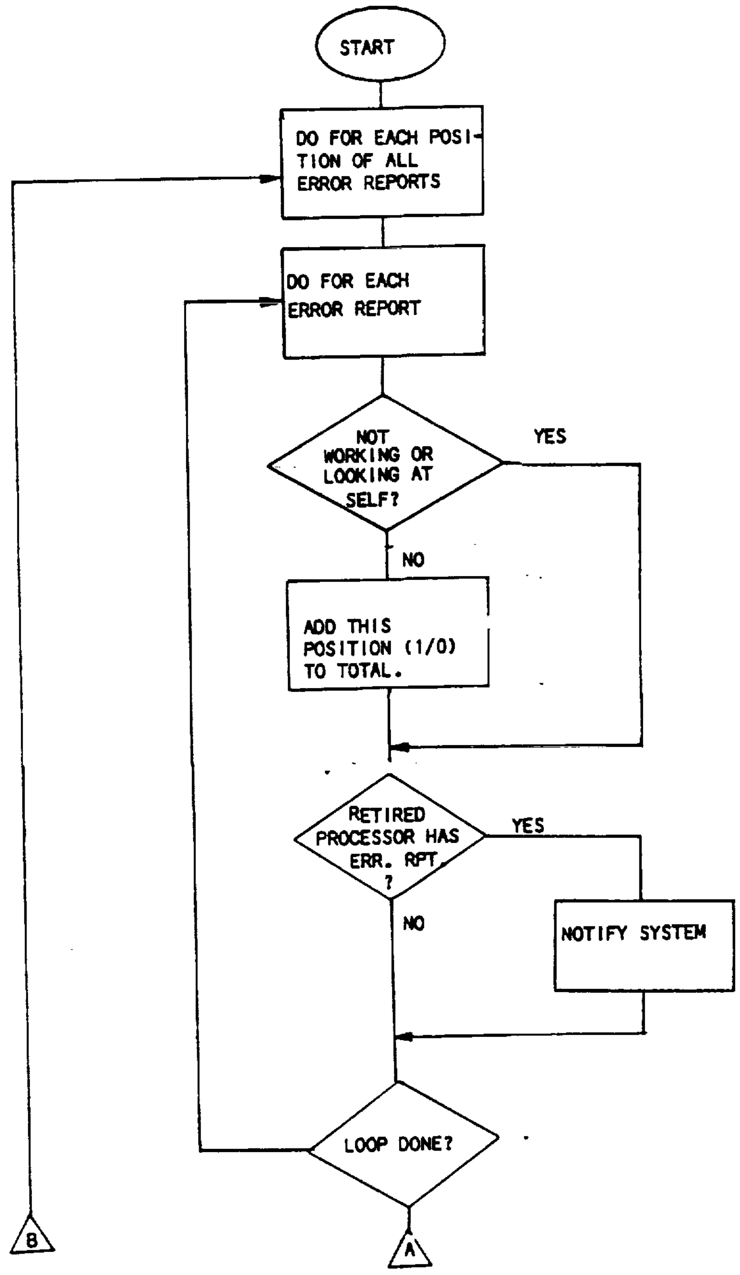
START
DO FOR EACH POSITION OF ALL ERROR REPORTS
DO FOR EACH ERROR REPORT
NOT WORKING OR LOOKING AT SELF?
YES
NO
ADD THIS POSITION (1/0) TO TOTAL.
RETIRED PROCESSOR HAS ERR. RPT. ?
YES
NO
NOTIFY SYSTEM
LOOP DONE?
B
A

Figure 3.2. Flow chart of Global Executive Acceptance Test

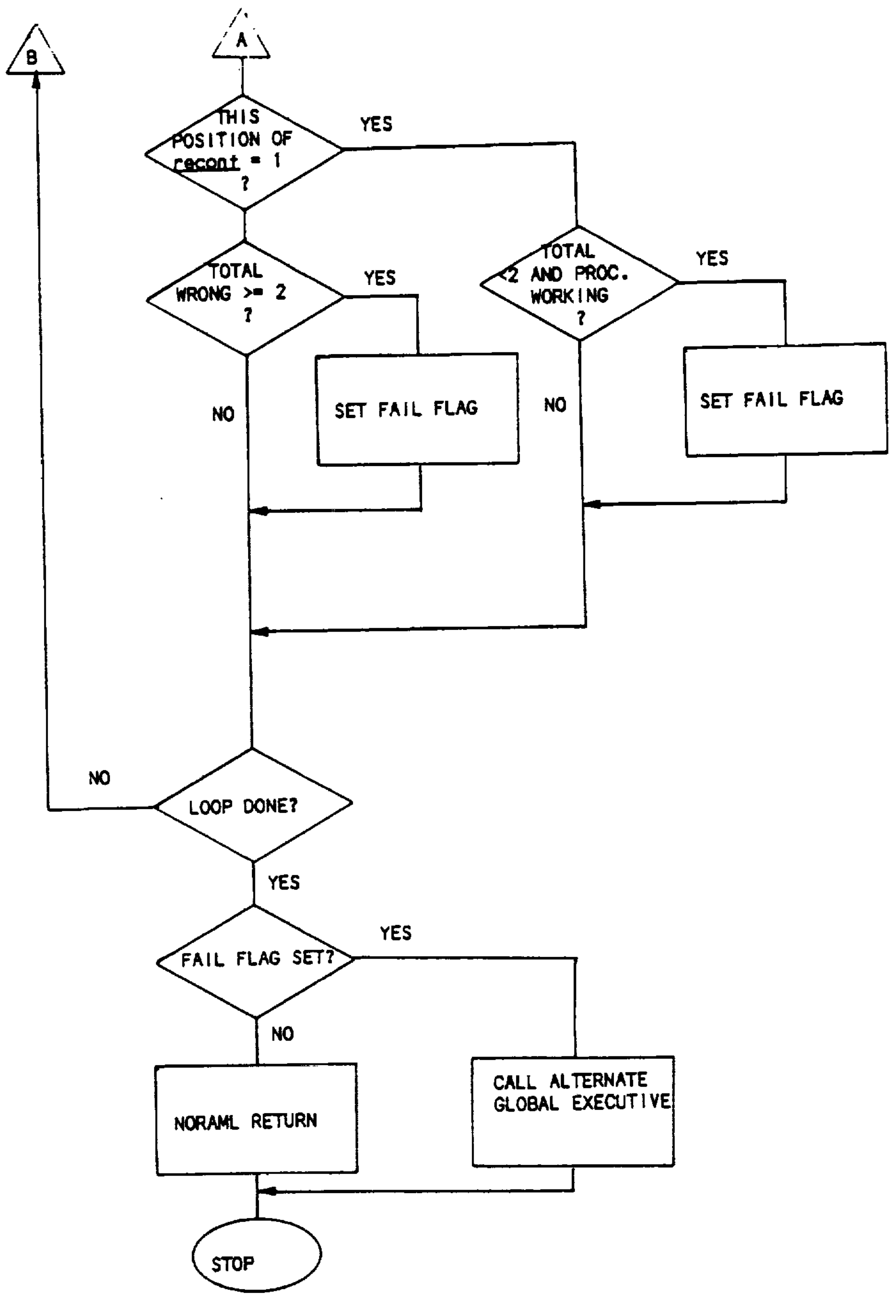

Figure 3.2 (continued). Flow chart of Global Executive Acceptance Test

before moving up to the next position. A second difference between the primary global executive and the acceptance test is the lack of an intermediate array (i.e. _procs_) for the storage of excess disagreements. Thus, once the number of discrepancies in a given position has been counted, it is immediately compared with the corresponding value in the reconfiguration word RECONF. If a discrepancy is detected, a flag is set that will result in the invocation of the alternate routine.

If a processor indicated as retired in the _working_ array is indicated as having excess disagreements in the input processor error reports, then one of three conditions exists (1) a processor marked for retirement is still a functioning part of the system, (2) an error exists which affects the state of the _working_ array, or (3) the error report(s) input to the global executive are not valid. Although the global executive can detect this discrepancy it cannot by itself isolate which of these three conditions caused the anomaly. As a result, the global executive acceptance test and alternate logic note the discrepancy to the system, but disregard the error reports in preparation of the _reconf_ word.

3.2. COVERAGE OF THE GLOBAL EXECUTIVE ACCEPTANCE TEST

The global executive acceptance test described above detects the following faults:

(1) failure to invoke the error reporter acceptance test

(2) failure to retire processors reported by at least two other processors as having an excess number of disagreements with the voter result, and

(3) marking for retirement processors which do not have an excess number of disagreements

Detection of the first fault occurs in the first module of the global executive acceptance test PREGEXEC. Two probable causes of the discrepancy are: (1) incorrect execution of the error reporter recovery block and (2) no invocation of the error reporter acceptance test. In either case, information reaching the global executive is suspect, and should be disregarded. If the rest of the system is properly functioning, the only penalty for no retirement at this point would be the unnecessary overhead necessitated by the higher number of active but not functional processors. Because the discrepancy is a processor disagreement from a majority vote, it should be counted in the total of the error reports of the other processors. If any other single disagreement occurs, the processor would be retired at the end of the next frame.

GEXECTEST detects both the second and third faults listed above. The number of processor disagreements registered in each processor error report are counted; retired or self-reporting processor disagreements are ignored. If the corresponding position in the reconfiguration word is zero when there are two or more reports which have bits set, or the reconfiguration word has a bit set when fewer than two (i.e. one or zero) processors are reported in the error words, then a boolean variable _failflg_ is set. The alternate global executive is invoked if _failflg_ is TRUE.

```
PROCEDURE PREGEXEC;
(*This procedure copies the least significant bits of the
 error reporter word bits into the most significant positions
  after checking the frame number *)

VAR
        excount:INTEGER;
        ERR:INTEGER;
        J,M:INTEGER;

begin
        mismatch:=0;                         (*mismatch is a global integer
                                               variable used for marking
                                               procs. not running ertask*)
        for j:=0 to maxprocessors do begin
                excount:=prevote[errerr,j] div 256;
                err:=prevote[errerr,j] mod 256;
                if excount=(framecount mod 256) then
                        prevote[errerr,j]:=257*err
                (*copy least sig. bits to most sig. position
                  if frame count OK*)
                else mismatch:=mismatch + 1;
                (*otherwise send word to error reporters in  subsequent
                  frame*)
                mismatch:=mismatch * 2;
        end;
end;
*
```

Figure 3.3. Listing for Global Executive Acceptance test; PREGEXEC

```
PROCEDURE GEXECTEST;
(*Global Executive Acceptance test*)

TYPE
        ZERO_ONE=0..1;
VAR
        DIVISOR,CHECK,I,J,SUM:INTEGER;
        FAILFLG:BOOLEAN;
        LAST_DIG:ZERO_ONE;
begin
        divisor:=1;
        failflg:=false;
        for i:=0 to maxprocessors do begin
                                (*...do for each position of report*)
(*This procedure is written under the assumption that the primary
   global executive has rotated the error reports a total of 8
  positions.  If this is not the case, additional division by
  (8 - 1 - maxprocessors)*2 for each error report is necessary *)
                sum:=0;
                for j:=0 to maxprocessors do begin
                                (*...do for each error report*)
                        last_dig:=(prevote[errerr,j] div divisor)
                                   mod 2;
                        if (not working[j]) or (i=j)
                          then last_dig:=0;
                        if(not working[i]) and (odd(last_dig))
                          then begin
                                recfail:=recfail+divisor;
                                (*recfail is a global integer
                                  showing a retired proc. working*)
                                last_dig:=0;
                          end;
                        sum:=sum + last_dig;
                end;
                check:=reconf div divisor;
                if odd(check)
                  then begin
                         if(sum<2)and(working[i]) then failflg:=true
                  end
                else if sum>=2 then failflg:=true;
                divisor:=divisor*2;
          end;
        if failflg then altgexec
         mismatch:=mismatch + 256*(framecount mod 256);
                (*indicate successful completion of acceptace test
                  to error reporters of next frame *)
end;
*
```

Figure 3.3 (continued). Listing of Global Executive Acceptance Test: GEXECTEST.

3.3. ALTERNATE GLOBAL EXECUTIVE

The alternate global executive, ALTGEXEC performs a function identical to the primary routine, but in an independent manner. The flow chart and listing for this procedure are shown in figures 3.4 and 3.5. Input to the alternate routine is the same as that used by the acceptance test: i.e. the error reports replicated by PREGEXEC. Unlike the primary routine, ALTGEXEC sums the totals of the disagreeing processors in descending order, and stores these totals in an integer array. If the totals in this array are less than two, then a zero is placed in the corresponding position of an alternate reconfiguration word, reconfa. Otherwise, the position is set to 1. A second difference between the primary and alternate is that the error words are not destructively read, and can be saved by the system if desired. As a final step of execution, ALTGEXEC sets the value of the primary reconfiguration word to that of the alternate. The primary reconfiguration word value can also be saved prior to execution of this step.

3.4. IMPLEMENTATION REQUIREMENTS

Three new procedures: GEXECTEST, PREGEXEC, AND ALTGEXEC are required for the operating system. PREGEXEC must be invoked prior to the execution of the primary global executive (GEXECTASK), and GEXECTEST is executed at its completion. This latter routine will invoke procedure ALTGEXEC, the alternate global executive, if required. Although the routines are presently declared as procedures, they may be changed to functions in order to be compatible with the form of GEXECTASK.

An additional global integer variable, called mismatch, is required. Frame count disceprancies detected in the PREGEXEC routine are recorded in a manner similar to processor error reports, i.e. by placing a "1" in the appropriate position of the word. The error reporters of other processor will read mismatch and increment the error counter for the appropriate processor if PREGEXEC reports a frame count disagreement.

A second global integer variable designated as recfail is used to enable the global executive to indicate the unsuccessful retirement of a failed processor. As is the case with mismatch, the faulty processor is noted by a "1" in the appropriate position. As noted previously, the global executive is not capable of determining whether the processor actually did not respond to the reconfiguration order for retirement or whether the "working" array is incorrect and thus, no further action can be taken by the global executive.

Changes in the values of each element of the prevote [errerr,*] array will occur due to the implementation of the fault-tolerant error reporter and global executive. As noted previously, PREGEXEC requires the frame count be encoded in the first half of the error report from each processor by the error reporter recovery block. In addition, the least significant bits of the error reports are replicated in the most significant positions by PREGEXEC. It is not anticipated that these changes have any impact on the rest of the SIFT executive.

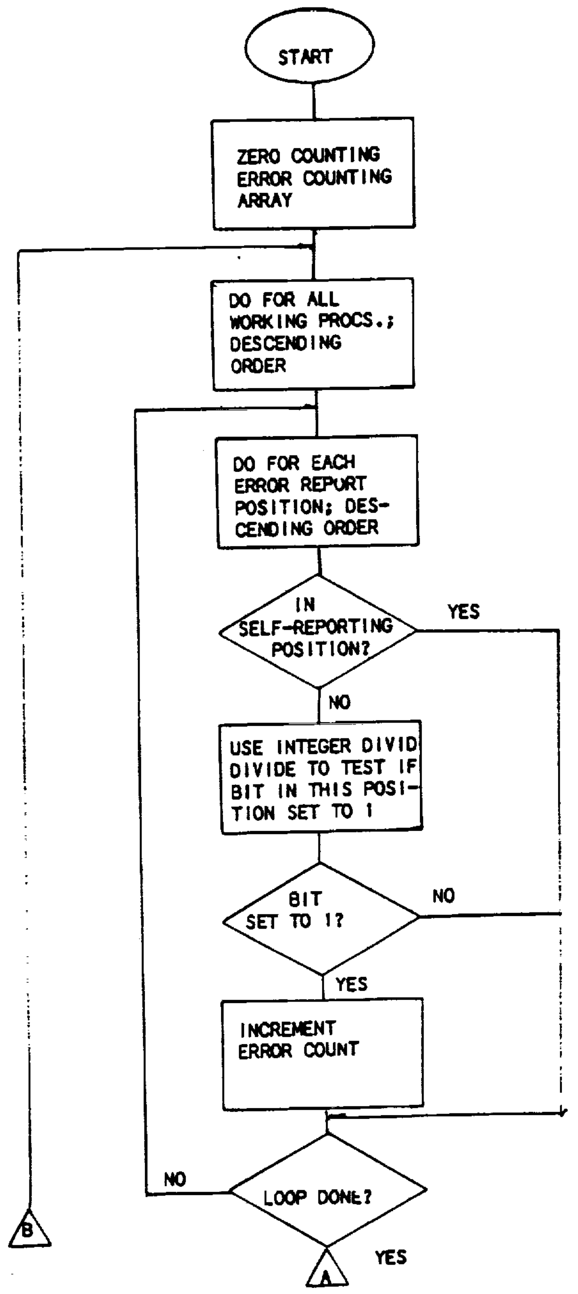
START
ZERO COUNTING ERROR COUNTING ARRAY
DO FOR ALL WORKING PROCS.; DESCENDING ORDER
DO FOR EACH ERROR REPORT POSITION; DESCENDING ORDER
IN SELF-REPORTING POSITION?
YES
NO
USE INTEGER DIVID DIVIDE TO TEST IF BIT IN THIS POSITION SET TO 1
BIT SET TO 1?
NO
YES
INCREMENT ERROR COUNT
LOOP DONE?
NO
YES
A
B

Figure 3.4. Flow chart of the Alternate Global Executive

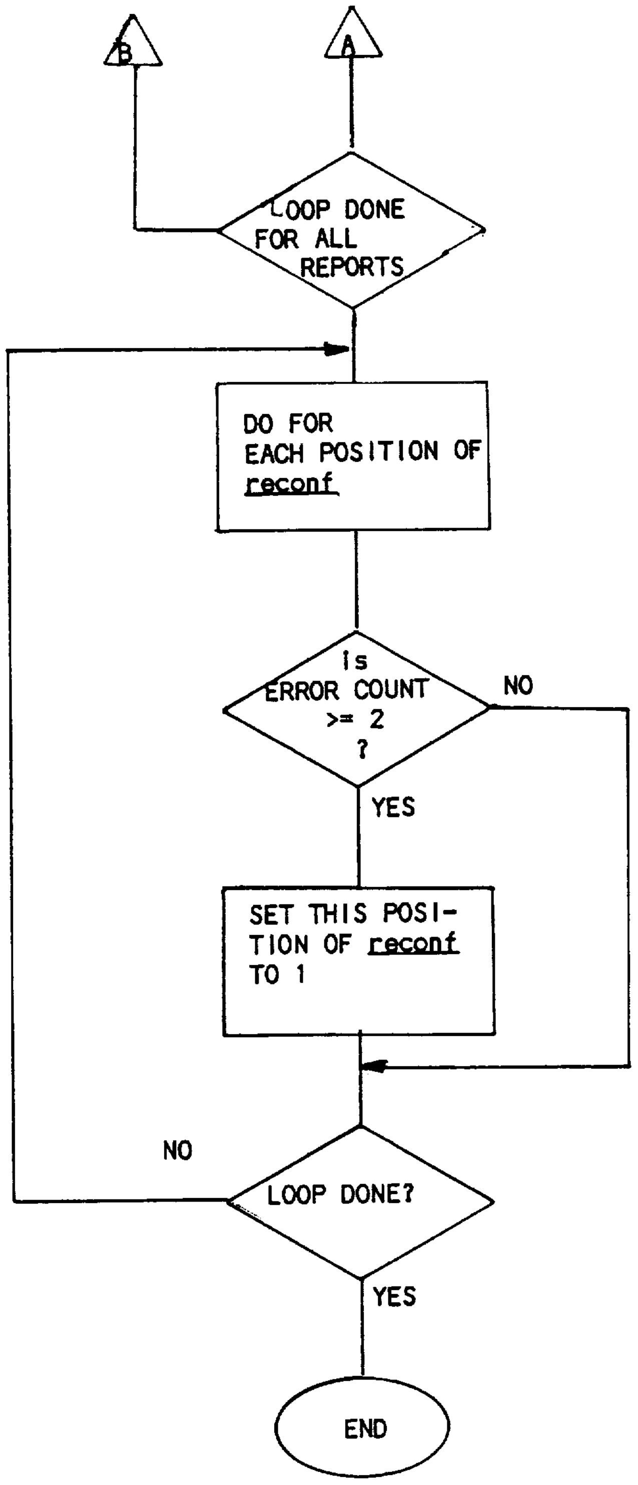
B
A
LOOP DONE FOR ALL REPORTS
DO FOR EACH POSITION OF reconf
Is ERROR COUNT >= 2 ?
NO
YES
SET THIS POSITION OF reconf TO 1
NO
LOOP DONE?
YES
END

Figure 3.4. (CONTINUED). Flow chart for ALTGEXEC.

```
PROCEDURE ALTGEXEC;
(*This is the alternate global executive*)

const   maxdiv=32;
VAR
        RECONFA,DIVISOR,MULT,J,K,L,M:INTEGER;
        ERCOUNT:PROCINT;
        LAST:INTEGER;
begin
        for j:=0 to maxprocessors do ercount[j]:=0;
                                (*...initialize ercount*)
        FOR J:= maxprocessors downto 0 do
          if working[j] then begin
                                (*...do for each error report*)
                divisor:=maxdiv;
                for k:=maxprocessors downto 0 do begin
                                (*...do for each position of report*)
                        if j=k then last:=0
                         else last:=prevote[errerr,j] div divisor;
                        if odd(last) then ercount[k]:=ercount[k]+1;
                        divisor:=divisor div 2;
                end
        end;
                                (*...now write reconfa*)
        reconfa :=0;
        mult:=1;
        for l:=0 to maxprocessors do begin
                if ercount[l]>=2 then reconfa:=reconfa+mult;
                mult:=mult*2;
        end;
        pre_broadcast(gexecreconf,reconfa);
end;
*
```

Figure 3.5. Listing of Alternate Global Executive

3.5. VALIDATION

The critical nature of the global executive acceptance test and the alternate global executive necessitates a comprehensive set of validation tests in order to demonstrate that the incorporation of these routines into the SIFT executive system do not negatively impact the overall reliability.

An exhaustive set of tests would involve testing each bit of each error report for the appropriate response for every possible configuration of all other error bits, the configuration of the working array, and the configuration of the reconf word. For six processors, there are a total of 281 trillion states of these variables, a rather intimidating number. However, the need for comprehenive testing remains. Thus, a major portion of the testing effort was devoted to the choice of an appropriate subset of these variables that would conclusively demonstrate that the global executive recovery block does not contain errors.

A fault-tree methodology was used to reduce the number of tests to a manageable number. The objective was to develop the trees to a sufficient level such that the primal events, i.e. those at the bottom of the tree, could be tested by a reasonable number of cases. If this testing showed that an insufficient number of primal events existed to make the top event (Failure of the global executive) true, then the validation would be complete.

The highest level tree is shown in figure 3.6. The top event, failure of the global executive recovery block, can be caused by either (1) a failure in the primary global executive and failure of the acceptance test to detect the failure or (2) the acceptance test invoking the alternate routine and failure of the alternate routine. For the purpose of this analysis, failure of the primary routine is a given, and thus, failures of the acceptance test and the alternate must be considered. However, because these routines function together as one unit, they are tested together in the validation procedure. Moreover, they both perform the same operation, i.e. determining the number of valid indications to discard a processor, and thus, are subject to the same types of faults. Hence, subsequent levels of development of these fault trees apply to both routines.

The next level of development shows the potential failed states of the acceptance test and the alternate global executive, which, as noted above, are the same. Two general classes of these possible failures exist: failure to identify a faulty processor (i.e. one where there are a sufficient number of agreeing error reports) in the reconf word, and failure to detect a "false positive" (i.e. the marking of a processor for retirement without the required number of agreeing error reports).

Figure 3.7 is the tree for the first class of failures: one, two or three faulty processors remaining unidentified. Validation test 1010 will test the software for each possible state of the error reports which would indicate a single processor as having failed. A large reduction in the number of states of the error reporter words can be achieved by consideration of the criteria for retirement: in order for a processor to be retired, the error reports of two other processors must indicate it had more than 2 disagreements from the majority in the previous frame. Thus, if the recovery block can be shown to detect any two working processors indicating a third processor as having failed then it will designate this failure if more than two processors so report.

As is shown in figure 3.8, failure to identify a single faulty processor may occur when 0, 1, 2, or 3 processors have been retired by the reconfiguration task. Considering all permutations of the SIFT configuration would lead to an impractical number of test cases, and the following logic describes the reduction in the validation process: there is only 1 SIFT configuration when all processors are working, and six possible configurations if a single processor is retired. These configurations are tested with all permutations of two processors indicating a third as faulty. Once the validations has establish that the global executive can correctly identify a failed processor with any single processor configured out of the system, validations for two processors configured out need only consider cases where the first retirement is held fixed (at processor 0) and the second is varied among the remaining 5. A similar line of reasoning can be used to consider three retired processors. Figure 3.9 shows the pattern of SIFT configurations that are tested for the single faulty processor case.

The next errors covered in this branch are the failure to identify two and three processors as having failed. In principle an exhaustive test should cover each possibility of two or three processors having failed. However, as implied in the fault tree, this can be broken into the failure to detect the first faulty processor, failure to detect the second, and failure to detect the third (if applicable). The failure to detect the first processor when no other processors have failed has been covered in test 1010A, along with arguments which extend the validity of this test to all states of <u>working</u> and <u>reconf</u>. This same argument can be easily extended to cover the case of more than one processor having failed.

Table 3.1 illustrates the validation tests required to cover all failure possibilities under the tree 1000. The validation procedure calls for processor 0 to be designated as faulty by processors 1 and 2, and that processors 1 though 5 be tested in turn in a manner similar to the single processor failed validation described above. An analagous line of reasoning can be used for the validation of the third processor failed case: processors 0 and 2 designate processor 1 as failed, processors 1 and 2 designate 0 as failed, and the third processor can be designated from the remaining processors (2 through 5). Table 3.1 lists this procedure explicitly.

Figure 3.10 shows the development of the class of errors concerned with designation of a functional processor as faulty. The possible failures resulting in a spurious processor failure indication include counting the error report of a processor which is not working as part of the total disagreement count, counting a processor's vote on itself, or the designation of a functional processor on the basis of 1 or no other processor error reports. These failures will be tested in tests designated as 1100A, 1100B, 1100C, and 1100D. A reduction in the number of tests to be performed occurs by the fact that these failures will take place for any value of <u>reconf</u>. Also, because the global executive acceptance test and alternate operate in the same statement sequence regardless of the SIFT state (i.e. there is no branching to different modules of the code depending on the values of <u>working</u>, <u>reconf</u>, or the failure of a particular processor), the same tests apply to <u>all</u> values of <u>working</u>.

Table 3.2 shows the list of validation tests and the range of <u>working</u>, <u>reconf</u>, and error reports. Tests 1100A and 1100B can be executed simultaneously with test 1000A. Test 1100C is executed by placing a single bit in all 36 possible error reporter positions, setting the corresponding position in reconf, and determining that both the acceptance test detects the error and the alternate routine functions correctly. Test 1100D is performed by setting each bit of reconf to 1 with no bits set in the error reporter words.

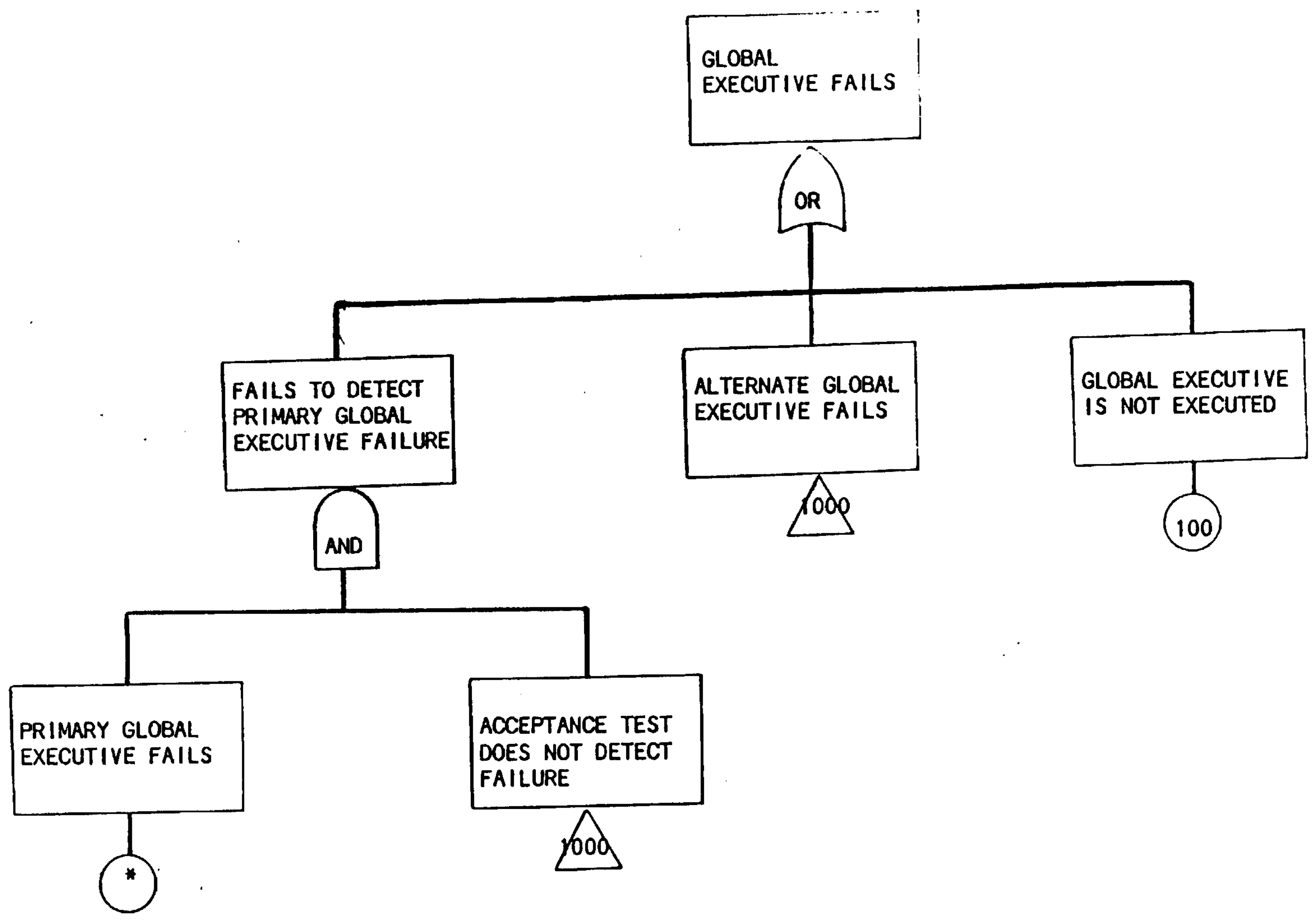
GLOBAL EXECUTIVE FAILS
OR
FAILS TO DETECT PRIMARY GLOBAL EXECUTIVE FAILURE
ALTERNATE GLOBAL EXECUTIVE FAILS
GLOBAL EXECUTIVE IS NOT EXECUTED
1000
100
AND
PRIMARY GLOBAL EXECUTIVE FAILS
ACCEPTANCE TEST DOES NOT DETECT FAILURE
1000
*

Figure 3.6. Top Level Fault Tree for Global Executive Validation

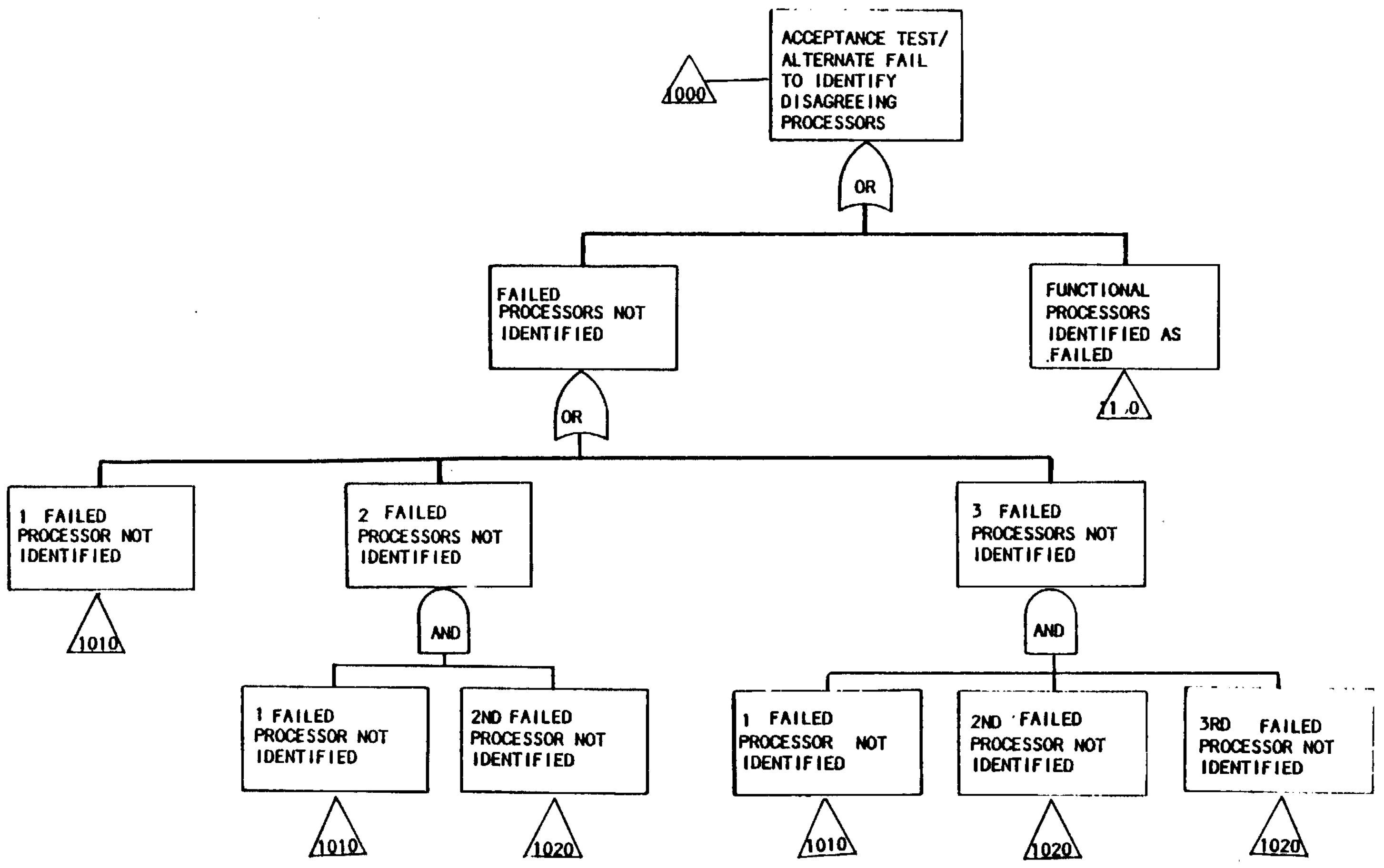
ACCEPTANCE TEST/ ALTERNATE FAIL TO IDENTIFY DISAGREEING PROCESSORS
1000
OR
FAILED PROCESSORS NOT IDENTIFIED
FUNCTIONAL PROCESSORS IDENTIFIED AS FAILED
11.0
OR
1 FAILED PROCESSOR NOT IDENTIFIED
1010
2 FAILED PROCESSORS NOT IDENTIFIED
AND
1 FAILED PROCESSOR NOT IDENTIFIED
1010
2ND FAILED PROCESSOR NOT IDENTIFIED
1020
3 FAILED PROCESSORS NOT IDENTIFIED
AND
1 FAILED PROCESSOR NOT IDENTIFIED
1010
2ND FAILED PROCESSOR NOT IDENTIFIED
1020
3RD FAILED PROCESSOR NOT IDENTIFIED
1020

Figure 3.7. Classes of Global Executive Faults

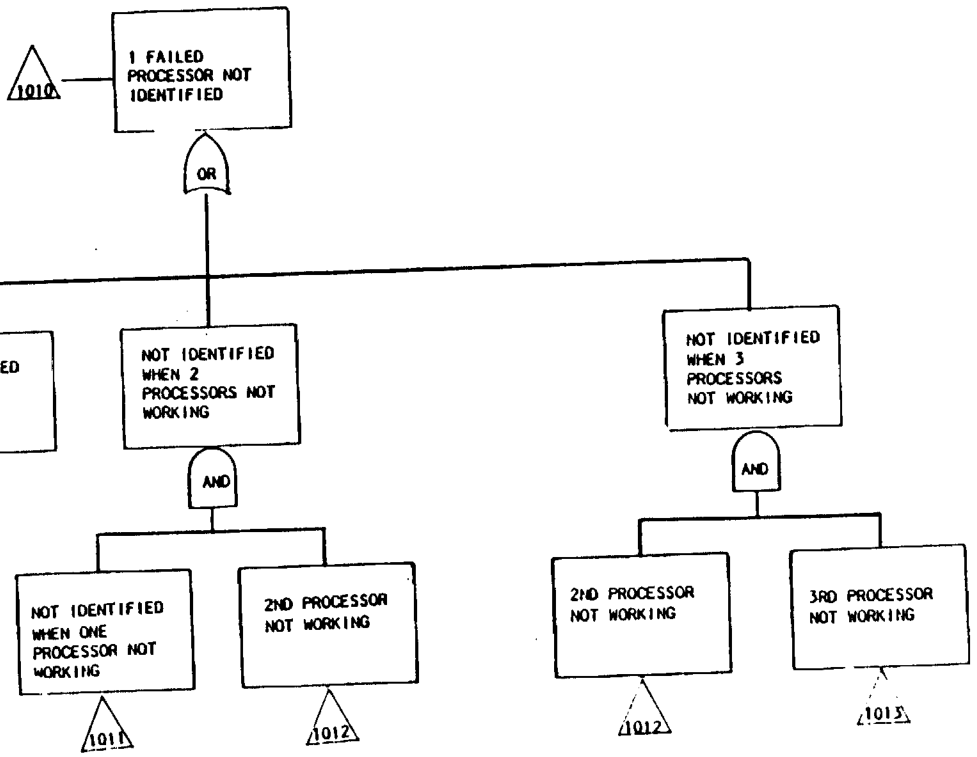
1 FAILED PROCESSOR NOT IDENTIFIED
1010
OR
NOT IDENTIFIED WHEN ALL PROCESSORS WORKING
1010
NOT IDENTIFIED WHEN ONE PROCESSOR NOT WORKING
1011
NOT IDENTIFIED WHEN 2 PROCESSORS NOT WORKING
AND
NOT IDENTIFIED WHEN ONE PROCESSOR NOT WORKING
1011
2ND PROCESSOR NOT WORKING
1012
NOT IDENTIFIED WHEN 3 PROCESSORS NOT WORKING
AND
2ND PROCESSOR NOT WORKING
1012
3RD PROCESSOR NOT WORKING
1013

Figure 3.8. Global Executive Detection Failure

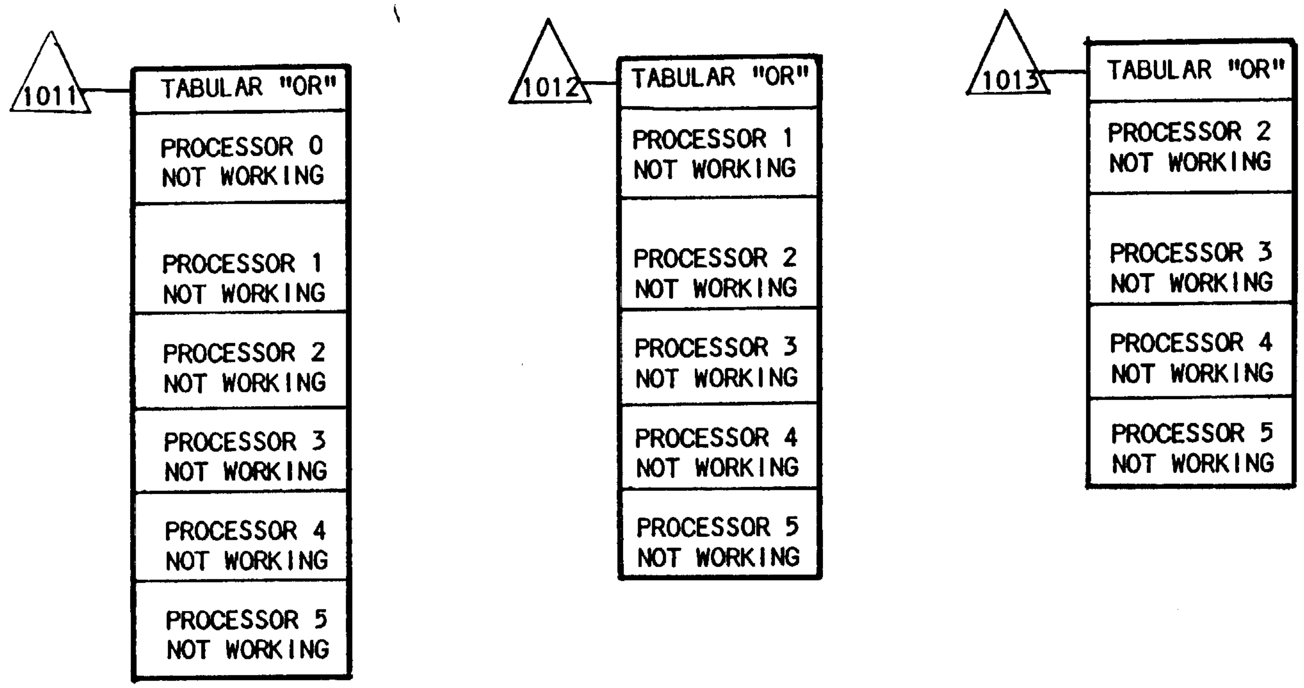

Figure 3.9. Expansion of Figure 3.7.

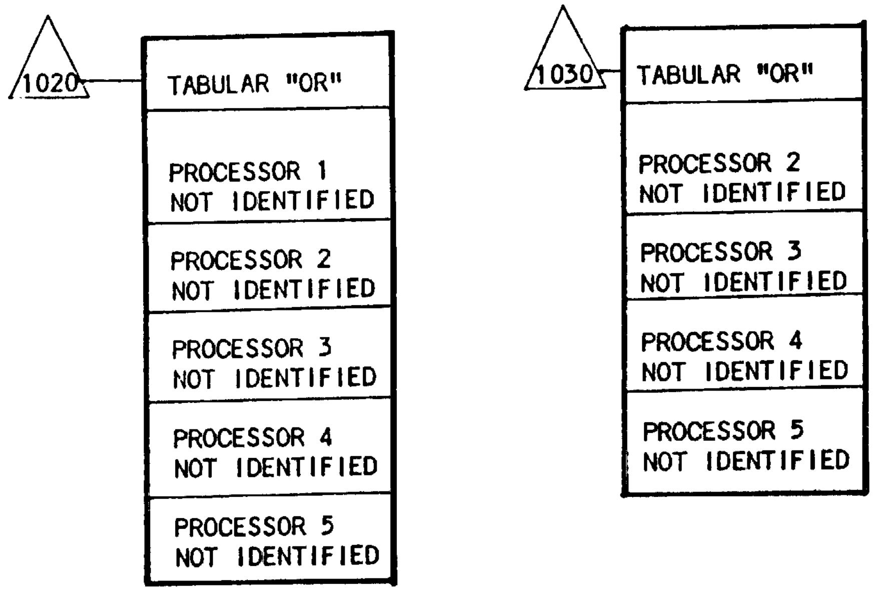
1020
TABULAR "OR"
PROCESSOR 1 NOT IDENTIFIED
PROCESSOR 2 NOT IDENTIFIED
PROCESSOR 3 NOT IDENTIFIED
PROCESSOR 4 NOT IDENTIFIED
PROCESSOR 5 NOT IDENTIFIED
1030
TABULAR "OR"
PROCESSOR 2 NOT IDENTIFIED
PROCESSOR 3 NOT IDENTIFIED
PROCESSOR 4 NOT IDENTIFIED
PROCESSOR 5 NOT IDENTIFIED

Figure 3.9. Expansion of Figure 3.7 (continued)

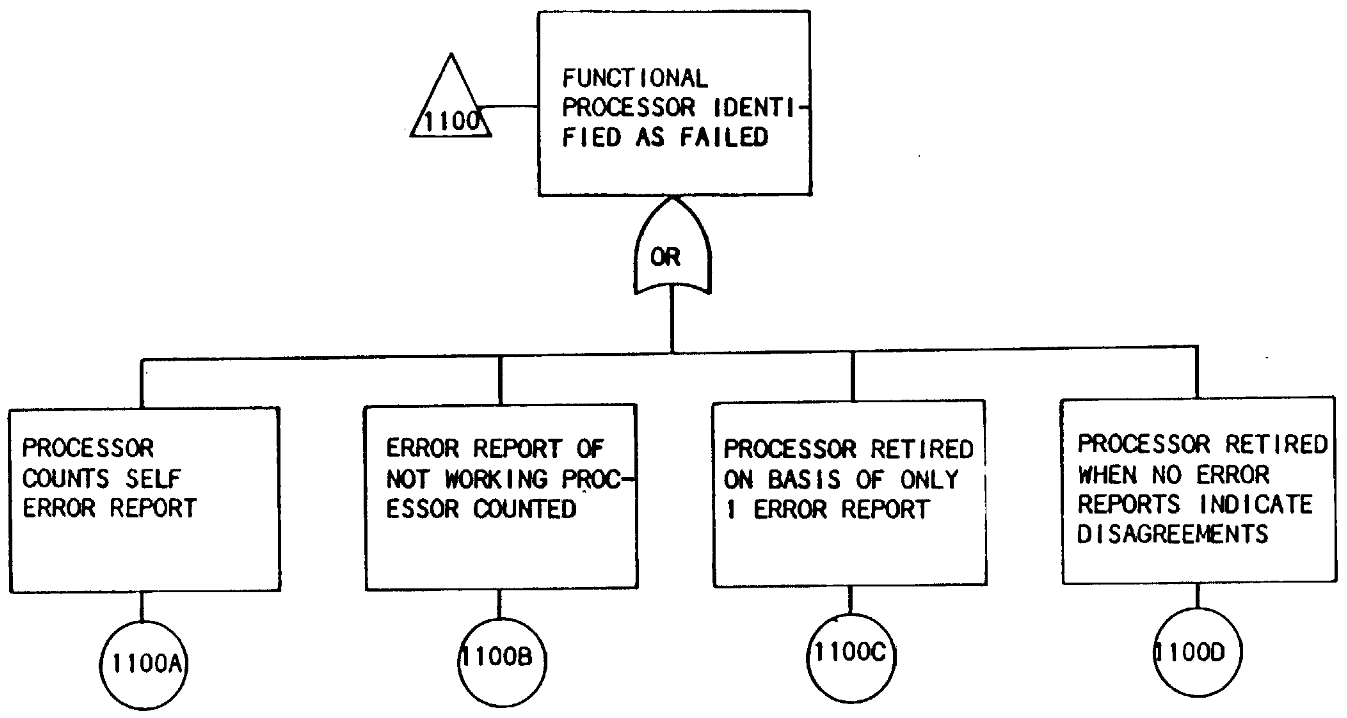
1100
FUNCTIONAL PROCESSOR IDENTIFIED AS FAILED
OR
PROCESSOR COUNTS SELF ERROR REPORT
1100A
ERROR REPORT OF NOT WORKING PROCESSOR COUNTED
1100B
PROCESSOR RETIRED ON BASIS OF ONLY 1 ERROR REPORT
1100C
PROCESSOR RETIRED WHEN NO ERROR REPORTS INDICATE DISAGREEMENTS
1100D

Figure 3.10. Spurious Identification of a Functional Processor

Table 3.1. Validation Tests for Global Executive Faulty Processor Dectection Failure

TEST	ERROR DESCR.	working	prevote	reconf	NOTE
1000A	1 FAILED PROC. UNDETECTED BY PRIMARY	0,1,2,3 not working	1 reported (1st indicated by any 2 other error reports)	0 retiring	1
1000B	2 FAILED PROC. UNDETECTED BY PRIMARY	0 not working	2 reported (2nd indicated by any 2 other error reports)	1 retiring (1st in any position of reconf)	2
1000C	3 FAILED PROC. UNDETECTED BY PRIMARY	0 not working	3 reported (3rd indicated by any 2 other error reports)	2 retiring (2nd in any position of reconf)	3

NOTES:

1. Failure of the primary global executive for this condition is manifested by both the following conditions: (1) one processor is identified as having excess disagreements by the individual error reports, and (2) the primary global executive did not mark this processor for retirement in the reconf word. This validation test is performed with 0,1,2,3 processors not working in order to determine whether the acceptance test and alternate are capable of detecting a single (or the first in the case of multiple) processor failure given any SIFT state. If any more than three processors are not working, the entire computer fails.

2. Failure of the primary global executive for this condition is manifested by the following conditions: (1) two processors are identified as having excess disagreements by the error reports, (2) the primary global executive marked the first processor for retirement in _reconf_, and (3) the primary global executive did _not_ mark the second processor for retirement. Validation testing for detection of the first processor given any configuration of _working_ with 0, 1, or 2 processors out and no processors marked for retirement has already been performed in 1C00A. Thus, this validation need only establish that the acceptance test can detect a second processor as having failed when the primary has marked only a single processor for retirement in _reconf_.

3. Failure of the primary global executive for this condition is manfested by the following conditions: (1) three processors are identified as having excess disagreements by the error reports, (2) the primary global executive marked the first two processors for retirement in _reconf_, and (3) the primary global executive did _not_ mark the third processor for retirement. Validation testing for detection of the first processor given any configuration of _working_ with 0, 1, or 2 processors out and no processors marked for retirement has already been performed in 1000A. Validation of the ability of the acceptance test to detect the second processor failure has been performed in 1000B. Thus, this validation need only establish that the acceptance test can detect a third processor as

having failed when the primary has marked only two processors for retirement in _reconf_.

Table 3.2. Validation Tests for Incorrect Retirement Errors of Global Executive

TEST	ERROR DESCR.	working	prevote	reconf
1100A	PROC. RETIRED ON BASIS OF SELF DIAGNOSIS	0,1,2,3 not working	1 other proc. reporting (any position)	1 proc. marked for retirement (any position)
1100B	PROC. RETIRED ON BASIS OF NOT WORKING PROC. REPORT	1 not working	1 other proc. reporting (any position)	1 proc. marked for retirement (any position)
1100C	PROC. RETIRED ON BASIS OF ONLY 1 ERROR REPORT	0 not working	1 other proc. reporting (any position)	1 proc. marked for retirement (any position)
1100D	PROC. RETIRED ON BASIS OF NO ERROR REPORTS	0 not working	no other proc. reporting	1 proc. marked for retirement (any position)

APPENDIX A. ERROR REPORTER DRIVER ROUTINES

Although both the error reporter acceptance test and the alternate routine are relatively brief procedures, a complete test required the simulation of a major portion of the SIFT operating system. The simulation program, called DRIVER, prepares the _errors_ and _working_ arrays of the voter and the _err_ word output of the error reporter based on externally input data. It next invokes the acceptance test, outputs its results to file TTY (for diagnostic purposes), and invokes the procedure if an error is detected. A complete listing of the program follows this description.

Figure A.1 is a heirachical representation of the program organization. The main program first invokes procedure IOFILES which either opens a previously written test data input file, prepares to write a new file, or simply accepts input and outputs directly to file TTY. Each of the subsequent procedures contain branches for the data source and destination defined in this routine. The main program the invokes procedure LIMREP, which determines the number of iterations (i.e. frames). FRAME COUNTER, the next procedure invoked, sets the value of _framecount_ against which _excount_, the internal counter of the error reporter, is compared. The program then invokes the VOTER and ERROR REPORTER procedures which, on the basis of input data, prepare the _working_ and _errors_ arrays and the _err_ and _excount_ variables. The ACCEPTANCE TEST procedure is then run, and the alternate error reporter is called by it in the event of the discrepancies discussed above. Subsequent iterations repeat the process from FRAME COUNTER through ACCEPTANCE TEST until the repetition limit is reached. Upon exiting the loop, the main program invokes procedure which closes any of the files opened in IOFILES and ends the simulation.

It should be noted that the actual error reporter acceptance test and alternate error reporter which were tested are shown in this listing, and that they are not identical to those shown in figures 2.2 and 2.3. These latter listing were changed to be compatible with the SIFT operating system (by including a _prebroadcast(errerr,err)_ statement) and eliminating display related statements (e.g. outputs to TTY and the BINPARS routine which represented the error words as binary numbers). An additional alteration was made to the acceptance test routine to include testing of the _mismatch_ variable. None of these changes are sufficiently significant to warrant additional validation testing.

Appendix C contains a sample output from this driver routine.

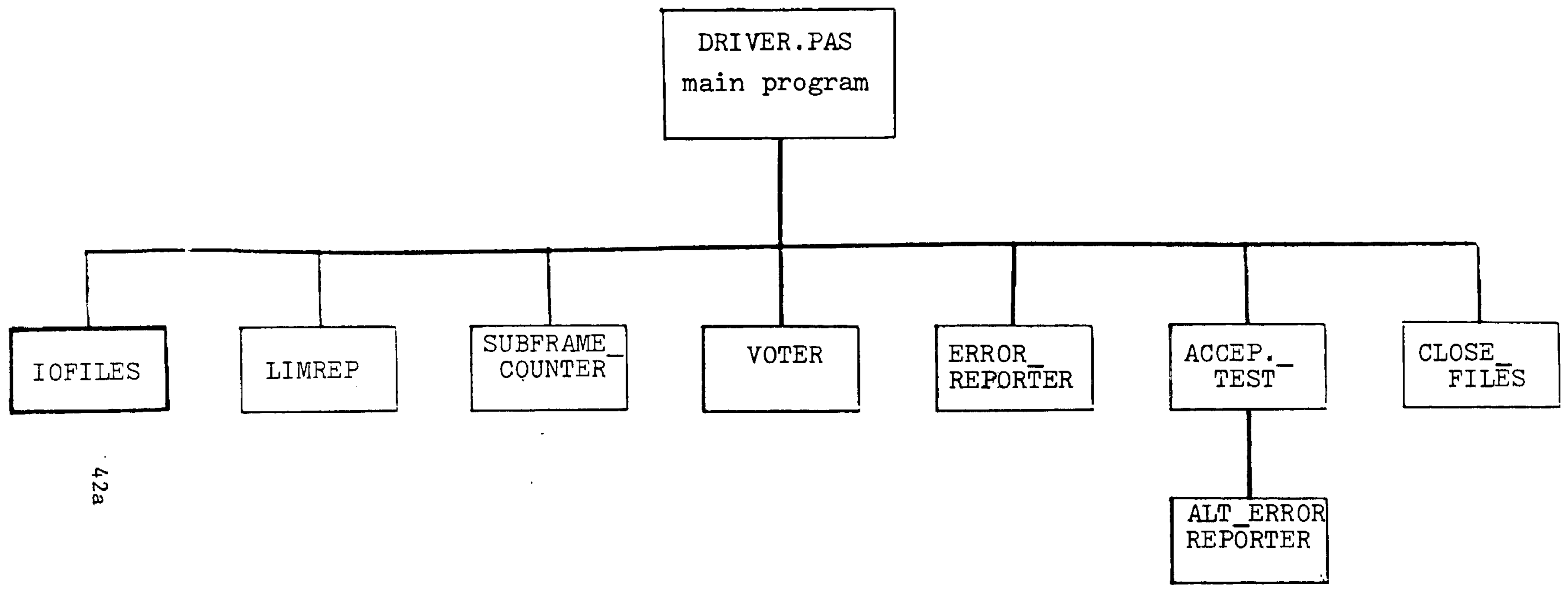

Figure A.1 Organization of Program DRIVER

```
PROGRAM DRIVER;
CONST
                        (*the following declarations are taken from
                          the AUGUST. 1980 VERSION OF THE SIFT
                          OPERATING SYSTEM *)
                    MAXPROCESSORS=5;
                    MAXframe=50;
                    THRESHOLD=2;
TYPE
          PROCESSOR=0..MAXPROCESSORS;
          PROCINT=ARRAY[PROCESSOR] OF INTEGER;
          PROCBOOL=ARRAY[PROCESSOR] OF BOOLEAN;
VAR
          ERR:INTEGER;
          ERRORS:PROCINT;
          REPORT:PROCINT;
          WORKING:PROCBOOL;
          framecount:INTEGER;
(*
                    the following declarations are necessary for
                    the error reporter recoverv block *)
          ERFAILS:integer;
(*
                    the following varables are necessry only
                    for the driver prcedures*)
          I,J,K:INTEGER;
          RPTLIM:INTEGER;
          FILENAME :PACKED ARRAY[1..8] OF CHAR;
          TITLE: PACKED ARRAY[1..40] OF CHAR;
          FIL:INTEGER;
          INTREP:PROCINT;
PROCEDURE IOFILES;
(*this program sets up files for both input and output
  as deterined by FIL input from the keyboard*)
begin
          writeln(tty,'Test of Error Reporter Recovery Block');
          writeln(tty,'I/O options: tty alone(0), input file(1)');
          writeln(tty,'Create File(2)');
          read(tty,fil);
          if fil>0 then begin
                    writeln(tty,'enter filename');
                    readln(tty);
                    readln(tty,filename);
                    if fil=1 then reset(input,filename)
                    else rewrite(output. filename);
                    writeln(tty,filename.' readv');
          end
          else writeln(tty,'I/O through terminal only');
end;

PROCEDURE LIMREP;
(*SET REPETITION LIMIT FOR MAN PROCEDURE*)
begin
          if fil<>1 then begin          (*promots for TY input*)
                    writeln(tty);
                    writeln(tty,'enter number of repetitions');
                    read(tty,rptlim);
                    if fil=2 then writeln(output,rptlim);
          end
          else begin
                    read(input,rptlim);
```

```
                   writeln(tty,rptlim, ' repetitions')
           end;
                   rptlim:=rptlim-1;
end;

PROCEDURE VOTER;
(*this procedure is to manaully input the error[p[i]]
  array generated in the voter routine*)

begin
   if fil<>1 then begin          (*tty iput*)
         writeln(tty,'procedure voter -- enter errors');
         for i:=0 to maxprocessors do begin
                         writeln(tty,'number of errors for processor ',
2);
                         read(tty,errors[i]);
                   writeln(tty,'working? (1/0)?');
                   read(tty,intrep[i]);
                   writeln(tty);
                   if fil=2 then write(output. errors[i],intrep[i]);
         end
   end
   else
         for i:=0 to maxprocessors do     (*file input*)
                   read(input,errors[i],intrep[i]);

   for i:=0 to maxprocessors do             (*all*)
         if intrep[i]<1 then working[i]:=false
          else working[i]:=true;
end;
PROCEDURE BINPARS(VAR NUM:INTEGER);
(*procedure to represent an integer as a 16 bit string *)
var
         binr: array[0..15] of integer;
         tnum:integer;
         divis,i,j:integer;
         byte: packed array[1..20] of char;
begin
         divis:=32768;
         if num>65535 then begin
         writeln(tty,'overflow');
                  num:=num mod 65535;
         end;
         tnum:=num;
         j:=0;
         for i:=15 downto 0 do begin
                  if tnum div divis>=1 then begin
                           tnum:=tnum mod divis;
                           binr[i]:=1
                  end
                  else binr[i]:=0;
                  divis:=divis div 2;
                  j:=j+1;
                  if binr[i] =1 then byte[j]:='1'
                           else byte[j]:='0';
                  if (i mod 4=0) then begin
                           j:=j+1;
                           bvte[j]:=' ';
                  end;
         end;
         writeln(tty,byte);
         writeln(tty);
```

```
end;
procedure error_reporter;
(*this procedure is to manually input the report[p[i]]
  array assumed to be generayted by the error reporter*)

VAR EXCOUNT:INTEGER;
begin
   if fil<>1 then begin              (*tty input*)
        writeln(tty);
                                     (*initialize the frame count *)
        writeln(tty,'framecount is',framecount:2,' enter execution');
        read(tty,excount);
                                     (*error reporter would be incrementing
                                       its own frae counter here *)
        writeln(tty,'title');
        readln(tty);
        readln(tty,title);
        writeln(tty);
        if fil=2 then write(output,excount,title);
        writeln(tty,'procedure error repoter -- enter report');
        err:=0;
        for i:= maxprocessors downto 0 do begin
                writeln(tty,'proc',i:2,'  err rpt.(1/0) =');
                read(tty,report[i]);
                err:=err*2;
                if (not working[i]) or (report[i]>0)
                 then err:=err+1;
                if fil=2 then write(output,report[i])
        end;
        writeln(tty);
        err:=err + 256*excount;  (*combine error and execution ct *)
        if fil=2 then write(output.err);
   end
   else begin
                                     (*file input *)
        read(input,excount,title);
        for i:=maxprocessors downto 0 do
                read(input.report[i]);
        read(input,err)
   end;
        writeln(tty);
        writeln(tty,title);
        writeln(tty,'frame no.',framecount:3,'execution',excount:3);
        writeln(tty);
        writeln(tty,'processor':15,'voter error':20,'error report':20,
                'working':20);
        for i:=0 to maxprocessors do begin
                writeln(tty);
                writeln(tty,i:10,errors[i]:20,report[i]:20,
                        intrep[i]:20);
        end;
        writeln(tty);
        writeln(tty,'primary error word= ',err:5);
        binpars(err);
end;
PROCEDURE FRAME_COUNTER;
(*This procedure is to simulate the execution counter on the
  error reporter acceptance test by means of manual input *)

        begin
*
```

```
                framecount:=framecount+1;
end;

PROCEDURE CLOSEFILES;
(*Close the input or output files if necessry*)
begin
        if fil=1 then close(input);
        if fil=2 then close (output);
end;

PROCEDURE ALT_ERROR_reportER;
(*this is the alterate error reporter*)

CONST
        ALLONES=377B;
VAR
        ERRA:INTEGER;  (*aternate error word*)
        I,K:INTEGER;

begin
        writeln(tty);
        writeln(tty,'alterate error reporter invoked');
        erra:=allones;
        k:=1;
        for i:=0 to maxprocessors do
                begin
                 if (errors[i]<threshold) and (working[i])
                  then erra:=erra-k;
                 k:=k*2;
        end;
        erra:=erra - (allones - k + 1);    (*remove leading bits*)
        writeln(tty,'alternate error word=',erra:5);
        err:=erra + 256*framecount;
        binpars(err);
        writeln(tty)
end;

PROCEDURE ACCEPTANCE_TEST;
(*error reporter accetance test*)
VAR
        EXCOUNT,WRONG,RIGHT,DIVISOR,CHECK,I,J:INTEGER;
        FAILFLG:BOOLEAN;

begin

        excount:= err div 256;
        err:=err mod 256;
    if excount=framecount then begin
        wrong:=0;
        failflg:=false;
        right:=0;
        divisor:=1;
        for j:=0 to maxprocessors do   (*check for omission errors*)
                begin
                if (errors[j]<threshold) and (working[j])
                 then right:=right+1;
                                        (*count for omissions test*)
                check:=err div divisor;
                                        (*shift err appropriate
                                          no. of places to the right*
*
```

```
p24100:30000
                    if odd(check) then begin
                            wrong:=wrong+1;  (*count for omissions test*)
                            if (errors[j]<threshold) and (working[j])
                             then failflg:=true (*check for false positives*
)
                    end;
                    divisor:=divisor*2;
            end;
            if wrong+right<>maxprocessors +1 then failflg:=true;
                                        (*omissions test*)
            if failflg then alt_error_reporter
            else writeln(tty,'error reporter OK');
    end
     else begin
            writeln(tty);
            writeln(tty,'primary error reporter did not run');
            alt error_reporter;
            writeln(tty);
      end;
end;

(*MAIN PROCEDURE*)

BEGIN
        IOFILES;
        LIMREP;
        REPEAT
                frame_COUNTER;
                VOTER;
                ERROR_REPORTER;
                ACCEPTANCE_TEST:
        UNTIL framecount>RPTLIM;
        CLOSEFILES;
END.
*
```

APPENDIX B. GLOBAL EXECUTIVE DRIVER ROUTINES

A significantly larger set of test cases was necessary for the global executive validation, and thus, its driver routine, GEXEC, used file input exclusively for the validation test input data. Two routines were used to input test data: INGEX, which accepted data directly from a terminal for generation of a small number of test cases, and MVTEST, which had an internal procedure for generation of a larger number of cases.

Program INGEX consists of 5 procedures: BINPARS, which represents integers as 16-bit binary numbers, CONV, which converts the input error reports and retiring processors into integers (err and reconf) used by the global executive, PRELIM, which opens a file for the test cases, OUTFILE, which writes the data to the file, and INDATA, which issues prompts to file TTY and processes the resultant input. The program first opens a file with procedure PRELIM, and then accepts input and writes to the file until the user specified number of test cases has been reached, and then saves the file for use by GEXEC.

MVTEST is composed of 4 procedures: ZERO, which zeros out the error reporter representation array for a new case, MVINIT which initializes an array containing all possible test cases for a given number of faulty processors, DISP, which performs additional processing and writes the cases to an output file, and MATCH, which selects a single test case from the possibilities generated by MV. Modifications to the main procedure, MATCH, and DISP were made for the generation of test cases for various configurations of the system (i.e. values of working) and number of processors becoming faulty in the current frame as described in section 3.5.

Program GEXEC contains 7 procedures: BINPARS, which was described above, PREGEXEC, the first module of the global executive acceptance test, ALTGEXEC, the alternate global executive, GEXECTEST, the second (and main) module of the acceptance test, INFILE, which reads files created by either INGEX or MVTEST, and PRELIM, which opens the files used by GEXEC. After PRELIM opens a file, the program flow is from INDATA, which prepares the input for the acceptance tests and alternate routine (if necessary), to PREGEXEC, GEXECTEST, and ALTGEXEC (if invoked by GEXECTEST). This sequence is repeated until the end of file condition is reached.

A modification of GEXEC, designated VALGEX, was used for creating a more terse output. This was necessitated by the large number of test cases (almost two thousand).

As was the case with the error reporter, modifications of the PREGEXEC, GEXECTEST, and ALTGEXEC procedures were made to remove all TTY I/O, make the output of the routines compatible with the SIFT operating system, and to include references to the mismatch variable described in sections 2 and 3. These minor alterations are not expected to affect the correctness of the routines as established by this validation.

Listings of INGEX, MVTEST, and GEXEC follow this description, and the output of GEXEC is described in Appendix C.

PROGRAM INGEX

```
PROGRAM INGEX;

CONST
        maxprocessors=5;
TYPE
        processor=0..maxprocessors;
        procint=array[processor] of integer;
VAR
        FILENAME: PACKED ARRAY[1..8] OF CHAR;
        CASENAME: PACKED ARRAY[1..40] OF CHAR;
        CASENO,MAXCASE,FRAMECOUNT:INTEGER;
        NUMREC.NUMOUT.NUMPROC,REPROC, FAULTPROC,
        NUMFAULT.PROCRET,PROCOUT:PROCESSOR;
        TVEC,INTREP,RETIRING:PROCINT;
        ERRORS:ARRAY[PROCESSOR] OF PROCINT;

PROCEDURE BINPARS(VAR NUM:INTEGER);
(*procedure to represent an integer as a 16 bit string *)
var
        binr: array[0..15] of integer;
        tnum:integer;
        divis,i.j:integer;
        bvte: packed array[1..20] of char;
begin
        divis:=32768;
        if num>65535 then begin
        writeln(tty,'overflow');
                num:=num mod 65535;
        end;
        tnum:=num;
        j:=0;
        for i:=15 downto 0 do begin
                if tnum div divis>=1 then begin
                        tnum:=tnum mod divis;
                        binr[i]:=1
                end
                else binr[i]:=0;
                divis:=divis div 2;
                j:=j+1;
                if binr[i] =1 then byte[j]:='1'
                        else byte[j]:='0';
                if (i mod 4=0) then begin
                        j:=j+1;
                        bvte[j]:=' ';
                end;
        end;
        writeln(tty);
        writeln(tty,byte);
        writeln(tty);
end;
FUNCTION CONV(ARAY:PROCINT):INTEGER;
VAR l,j,k:integer;
begin
        j:=1;
        k:=0;
        for l:=0 to maxprocessors do begin
                k:=k+aray[l]*j;
                j:=j*2;
*
```

```
        end;
        conv:=k;
end;

PROCEDURE PRELIM;
begin
        writeln(tty,'Enter file name');
        readln(tty);
        read(tty,filename);
        writeln(tty,'Enter total number of processors');
        read(tty,numproc);
        writeln(tty,'Enter number of cases');
        read(tty,maxcase);
end;

PROCEDURE OUTFILE;
(*write output to file and report to tty*)
VAR prevote,j,k,reconf,numfault:integer;
begin
        writeln(output.casename);
        writeln(tty,'case ',casename);
        writeln(tty);
        for k:=0 to maxprocessors do write(output.intrep[k]);
        writeln(tty,'working status');
        for k:=0 to maxprocessors do write(tty,intrep[k]);
        writeln(tty);
        for k:=0 to maxprocessors do begin
                for j:=0 to maxprocessors do tvec[j]:=errors[k,j];
                prevote:=conv(tvec) + 256*framecount;
                writeln(tty,'error report for processor ',k:2);
                binpars(prevote);
                write(output.prevote);
        end;
        reconf:=conv(retiring) + 256*framecount;
        writeln(tty,'Reconfiguration word');
        binpars(reconf);
        writeln(output,reconf);
end;

PROCEDURE INDATA;
(*This procedure does the actual test case input*)
VAR i,m,n,j:integer;
begin
        writeln(tty,'enter case name');
        readln(tty);
        read(tty,casename);
        writeln(tty,'Enter framecount');
        read(tty,framecount);
        for m:=0 to maxprocessors do begin
                intrep[m]:=0;
                retiring[m]:=0;
                for n:=0 to maxprocessors do errors[m,n]:=0;
        end;
                        (*...Prepare the intrep array*)
        writeln(tty,'How many processors are not working?');
        read(tty,numout);
        if numout>0 then begin
                writeln(tty,'which processors not working?');
                for i:=1 to numout do begin
                        read(tty,procout);
*
```

```
                        intrep[procout]:=1;
                end;
        end;
                        (*...Prepare the errors array*)
        writeln(tty,'How many processors are faulty?');
        j:=0;
        readln(tty,numfault);
        if numfault>0 then repeat
                j:=j+1;
                writeln(tty,'wrong processor', j:3);
                writeln(tty,'which processor is faulty?');
                read(tty,faultproc);
                writeln(tty,'how many processors report it as faulty?');
                read(tty,numproc);
                writeln(tty,'which processors reported it?');
                for i:=1 to numproc do begin
                        read(tty,reproc);
                        errors[reproc,faultproc]:=1;
                end
        until j=numfault;
        writeln(tty);
        writeln(tty,'Summary of Error Reports of all processors');
        writeln(tty,'Reporting        Faulty');
        writeln(tty,'processors       processors');
        writeln(tty);
        for i:=0 to maxprocessors do begin
                writeln(tty);
                write(tty,i:3,'            ');
                for m:=0 to maxprocessors do
                        write(tty,errors[i,m]:3);
        end;
                        (*...Prepare the reconf word*)
        writeln(tty);
        writeln(tty);
        writeln(tty,'How many processors are reconfigured out?');
        read(tty,numrec);
        if numrec>0 then begin
                writeln(tty,'which processors are reconfigured out?');
                for i:=1 to numrec do begin
*
```

```
                        for i:=1 to numrec do begin
                                read(tty,procret);
                                retiring[procret]:=1;
                        end;
                end;
        end;

        (*MAIN PROCEDURE*)
        begin
                prelim;
                if numproc=maxprocessors then begin
                        rewrite(output,filename);
                        for caseno:=1 to maxcase do begin
                                indata;
                                outfile;
                        end;
                        close(output);
                end
                else writeln(tty,'change maxprocessors, current value is',
                                maxprocessors);
        end.
*
```

PROGRAM MVTEST

```
PROGRAM MVTEST;
CONST MAXPROCESSORS=5;
      maxv=14;
VAR
        kount,kw:integer;
        reconf:integer;
        filename: packed array[1..8] of char;
        working: array[0..maxprocessors] of integer;
        MV:ARRAY[0..MAXPROCESSORS,0..MAXV] OF INTEGER;
                (* THE MV ARRAY COLUMNS WILL BE USED IN E TO
                   FORM THE DIFFERENT COMBINATIONS OF ERROR
                   REPORTS REQUIRED FOR THE VALIDATION *)
        E:ARRAY[0..MAXPROCESSORS,0..MAXPROCESSORS] OF INTEGER;
                (* E IS THE ARRAY REPRESENTING ERROR REPORTS *)
        A:ARRAY[0..1,0..MAXV] OF INTEGER;
                (* A IS THE ARRAY FOR MARKING WHICH PROCS REPORT *)
PROCEDURE ZERO;
  (* zero the e array *)
VAR I,J:INTEGER;
begin
        for i:=0 to maxprocessors do
                for j:=0 to maxprocessors do
                        e[i.j]:=0;
end;
PROCEDURE MVINIT;
  (* initialize the mv and associated a arrays *)
VAR I,J,K,L,M:INTEGER;
begin
        for l:=0 to maxprocessors do
                for m:=0 to maxv do mv[l.m]:=0;
        j:=0;
        for i:=0 to maxprocessors - 1 do
                for k:=i+1 to maxprocessors do begin
                        mv[i.j]:=1;
                        mv[k,j]:=1;
                        A[0,J]:=I:
                        A[1,J]:=K;
                        j:=j+1;
                end;
        for l:=0 to maxprocessors do working[l]:=0;
 (*
        working[0]:=1;
        working[1]:=1;
        working[kw]:=1;
 *)
END;
PROCEDURE DISP(VAR L.J:INTEGER);
  (* write the output file for use by GEXEC *)
VAR I,K,M,S:INTEGER;
begin
        kount:=kount+1;
        writeln(output,'proc',j:2,' outvoted; procs',
                a[0,l]:2,a[1,l]:2,' reporting. proc. 0 failure report
);
        for i:=0 to maxprocessors do write(output,working[i]);
        for i:=0 to maxprocessors do begin
                m:=1;
                s:=0;
                for k:=0 to maxprocessors do begin
                        s:=e[i.k]*m+s;
                        m:=m*2;
                end;
                s:=s + 256*kount;
                write(output.s);
        end;
        reconf:=reconf + 256*kount;
                (*...for more than 1 proc. out, reconf should have
                      constants added to it:
                        1 - for one proc. out
                        3 - for two procs. out
                        7 - for 3 procs. out  *)
        writeln(output,reconf);
end;
PROCEDURE MATCH;
VAR I,J,K,L,M:INTEGER;
begin
        for l:=0 to maxv do      (*.. l is col. of mv*)
                for j:=0 to maxprocessors do begin
                   zero;
                   for i:=0 to maxprocessors do
                        e[i.j]:=mv[i.l];
        (* mark procs 0 and 1 excess disagreements here*)
                e[4,0]:=1;
                e[5,0]:=1;
                e[4,2]:=1;
                e[5,2]:=1;
                e[4,1]:=1;
                e[5,1]:=1;
                   disp(L,J);
                   end;
end;
(*MAIN PROCEDURE*)
BEGIN
        writeln(tty,'2 processors tet, enter filename');
        readln(tty);
        read(tty,filename);
        rewrite(output,filename);
        kount:=0;
        writeln(tty,'enter reconf');
        read(tty,reconf);
 (*
        writeln(tty,'which additional. proc. out?');
        read(tty,kw);
 *)
        MVINIT;
        MATCH;
        close(output);
        writeln(tty,'file complete');
END.
*
```

PROGRAM GEXEC

```
PROGRAM GEXEC;
(*This is the set of routines associated with the global
  executive *)

CONST
          MAXPROCESSORS=5;
          maxsubframe=50;
          threshold=2;
          maxbufs=1;
          errerr=1;
TYPE
          processor=0..maxprocessors;
          procint=array[processor] of integer;
          procbool=array[processor] of boolean;
          buffer=0..maxbufs;
VAR
          WORKING:PROCBOOL;
          FRAMECOUNT,CASENO:INTEGER;
          PREVOTE:ARRAY[BUFFER] OF PROCINT;
          RECONF:INTEGER;


PROCEDURE BINPARS(VAR NUM:INTEGER);
(*procedure to represent an integer as a 16 bit string *)
var
          binr: array[0..15] of integer;
          tnum:integer;
          divis,i,j:integer;
          byte: packed array[1..20] of char;
begin
          divis:=32768;
          if num>65535 then begin
          writeln(tty,'overflow');
                    num:=num mod 65535;
          end;
          tnum:=num;
          j:=0;
          for i:=15 downto 0 do begin
                    if tnum div divis>=1 then begin
                              tnum:=tnum mod divis;
                              binr[i]:=1
                    end
                    else binr[i]:=0;
                    divis:=divis div 2;
                    j:=j+1;
                    if binr[i] =1 then byte[j]:='1'
                              else byte[j]:='0';
                    if (i mod 4=0) then begin
                              j:=j+1;
                              byte[j]:=' ';
                    end;
          end;
          writeln(tty,byte);
          writeln(tty);
*
```

```
end;
PROCEDURE PREGEXEC;
(*This procedure copies the least significant bits of the
 error reporter word bits into the most significant positions
  after checking the frame number *)

VAR
        excount:INTEGER;
        ERR:INTEGER;
        J,M:INTEGER;

begin
        for j:=0 to maxprocessors do begin
                excount:=prevote[errerr,j] div 256;
                err:=prevote[errerr,j] mod 256;
                if excount=framecount then
                        prevote[errerr,j]:=257*err
                else writeln(tty,'processor',j:3,' excount mismatch');
        end;
end;

PROCEDURE ALTGEXEC;
(*This is the alternate global executive*)

const   maxdiv=32;
VAR
        RECONFA,DIVISOR,MULT,J,K,L,M:INTEGER;
        ERCOUNT:PROCINT;
        LAST:INTEGER;
begin
        for j:=0 to maxprocessors do ercount[j]:=0;
                                (*...initialize ercount*)
        FOR J:= maxprocessors downto 0 do
          if working[j] then begin
                                (*...do for each error report*)
                divisor:=maxdiv;
                for k:=maxprocessors downto 0 do begin
                                (*...do for each position of report*)
                        if j=k then last:=0
                         else last:=prevote[errerr,j] div divisor;
                        if odd(last) then ercount[k]:=ercount[k]+1 ;
                        divisor:=divisor div 2;
                end
        end;
                                (*...now write reconfa*)
        reconfa :=0;
        mult:=1;
        for l:=0 to maxprocessors do begin
                if ercount[l]>=2 then reconfa:=reconfa+mult;
                mult:=mult*2;
        end;
        writeln(tty,'alternate reconf word');
        binpars(reconfa);
end;
*
```

```

PROCEDURE GEXECTEST;
(*Global Executive Acceptance test*)

TYPE
        ZERO_ONE=0..1;
VAR
        DIVISOR,CHECK,I,J,SUM:INTEGER;
        FAILFLG:BOOLEAN;
        LAST_DIG:ZERO_ONE;
begin
        divisor:=1;
        failflg:=false;
        for i:=0 to maxprocessors do begin
                                (*...do for each position of report*)
        (*implement error word shifts here*)
                sum:=0;
                for j:=0 to maxprocessors do begin
                                (*...do for each error report*)
                        last_dig:=(prevote[errerr,j] div divisor)
                                mod 2;
                        if (not working[i]) or (i=j)
                          then last_dig:=0;
                        sum:=sum + last_dig;
                end;
                check:=reconf div divisor;
                if odd(check) then begin
                        if(sum<2)and(working[i]) then failflg:=true
                end
                else if sum>=2 then failflg:=true;
                divisor:=divisor*2;
        end;
        if failflg then altgexec
        else writeln(tty,'global Executive OK');
end;
PROCEDURE INFILE;
(*Read data from file input after main procedure has opened it*)
var
        casename:packed array[1..40] of char;
        intrep:procint;
        k:integer;

begin
        readln(input,casename);
        for k:=0 to maxprocessors do read(input,intrep[k]);
        for k:=0 to maxprocessors do read(input,prevote[errerr,k]);
        readln(input,reconf);
        writeln(tty);
        writeln(tty);
        writeln(tty,casename);
        write(tty,'Case',caseno:3,'  Enter framecount ');
        read(tty,framecount);
        writeln(tty);
        writeln(tty,'Failed processors');
*
```

```
          for k:=0 to maxprocessors do begin
                    write(tty,intrep[k]:3);
                    if intrep[k]=1 then working[k]:=false
                     else working[k]:=true;
          end;
          writeln(tty);
          for k:=0 to maxprocessors do begin
                    writeln(tty,'error report for processor',k:3);
                    binpars(prevote[errerr,k]);
          end;
          writeln(tty,'Reconfiguration Word');
          binpars(reconf);
end;

PROCEDURE PRELIM;
(*Initial prompts and opening of data file*)
var filename:packed array[1..8] of char;
begin
          writeln(tty,'Global Executive Recovery Block Driver --');
          writeln(tty,'Enter Data File');
          readln(tty);
          read(tty,filename);
          reset(input,filename);
end;

(*MAIN PROCEDURE*)
begin
          prelim;
          caseno:=0;
          while not eof(input) do begin
                    caseno:=caseno+1;
                    infile;
                    pregexec;
                    gexectest
          end;
          writeln(tty,'Tests Complete');
          close(input);
end.
*
```

APPENDIX C. DEMONSTRATION OF VALIDATION PROCEDURES

This appendix contains examples of output which demonstrate the manner in which the fault-tolerant software for the error reporter and the global executive were validated. Section C.1. describes the output from DRIVER used to demonstrate the correctness of the error reporter, and section C.2 describes the GEXEC output which showed the proper operation of the fault tolerant global executive.

C.1. Error Reporter Validation

Figure C.1 is the output generated by the DRIVER program using data for 1 processor out. A total of five "frames" (i.e. test cases) are shown. The first line is the abbreviated title "proc 1 exc undtctd err", which is the designation for processor no. 1 having an excess number of errors undetected by the primary error reporter. The next line shows the value of framecount and excount (which were taken to be the same for the cases shown here). The next item on the output is a table showing the number of errors counted by the voter, the error reporter output (0 = no excess disagreements, 1 = excess disagreements), and the working status (0 = not working, 1 = working) for each of the six processors. The following line shows the integer value of the error word including the frame count encoded in the 8 most significant bits, and immediately below it is the binary representation produced by procedure BINPARS (see appendix B).

Because the primary error report (contained in the file) was incorrect, the error reporter acceptance test invoked the alternate, which generated an error report whose integer value (not including the frame count) is shown on the next line and whose binary representation (including the frame count) is shown immediately below.

This particular case demonstrates that the acceptance test can detect failure of the primary error reporter to note an excess number of disagreements in processor 1 when no other processors have failed and when all are working. Succeeding cases shown in this output demonstrate that failure of the primary routine to detect excess disagreements for processors 2, 3, 4, and 5 when no processors have been retired or have become faulty in this frame. The entire validation sequence described in section 2.5 consists of performing a sequence similar to this for processors 0 through 6 when 1, 2, or 3 additional processors become faulty in the current frame and when 1, 2, or 3 other processors have been retired. Although these validations were performed, they are not included in this report for the sake of brevity.

```
err1       ready
             5 repetitions

proc 1 exc undtctd errr
frame no.  1execution  1

       processor           voter error          error report              working

          0                    0                    0                        1

          1                    3                    0                        1

          2                    0                    0                        1

          3                    .                    0                        1

          4                    0                    0                        1

          5                    0                    0                        1

primary error word=  256
0000 0001 0000 0000


alterate error reporter invoked
alternate error word=     2
0000 0001 0000 0010



proc 2 exc undtctd err
frame no.  2execution  2

       processor           voter error          error report              working

          0                    0                    0                        1

          1                    0                    0                        1

          2                    5                    0                        1

          3                    0                    0                        1

          4                    0                    0                        1

          5                    0                    0                        1

primary error word=  512
0000 0010 0000 0000


alterate error reporter invoked
alternate error word=     4
0000 0010 0000 0100
```

FIGURE C.1. Error Reporter Validation Output

```
proc 3 exc undt ctd disgr
frame no.  3execution  3

      processor           voter error         error report          working

         0                    0                    0                   1

         1                    0                    0                   1

         2                    0                    0                   1

         3                    4                    0                   1

         4                    0                    0                   1

         5                    0                    0                   1

primary error word=  768
0000 0011 0000 0000


alterate error reporter invoked
alternate error word=    8
0000 0011 0000 1000



proc 4 exc undt ctd err
frame no.  4execution  4

      processor           voter error         error report          working

         0                    0                    0                   1

         1                    0                    0                   1

         2                    0                    0                   1

         3                    0                    0                   1

         4                    6                    0                   1

         5                    0                    0                   1

primary error word= 1024
0000 0100 0000 0000


alterate error reporter invoked
alternate error word=   16
0000 0100 0001 0000
```

Figure C.1. (continued) Error Reporter Validation Output

```
proc 5 exc undt ctd err
frame no. 5execution 5

      processor         voter error        error report          working

          0                  0                  0                    1

          1                  0                  0                    1

          2                  0                  0                    1

          3                  0                  0                    1

          4                  0                  0                    1

          5                  3                  0                    1

primary error word= 1280
0000 0101 0 000 0000


alterate error reporter invoked
alternate error word=   32
0 000 0101 0 010 0000
```

Figure C.1. (continued) Error Reporter Validation Output

C.2. Global Executive

Figure C.2 shows an excerpt from the output generated by program GEXEC. Two cases are shown from an MVTEST generated file containing cases in which processor 1 is marked as having failed by processors 5 and 6, and 1 additional processor is marked for retirement in the reconfiguration word by the primary global executive. The first line shows the title of the case, i.e. "proc 0 outvoted; procs 0, 1 reporting" . Thus, processor 0 is marked as having excess disagreements by processors 0 and 1, and processor 1 is indicated as having excess disagreements in the error reports of processors 5 and 6. The second line of the output is the frame count check, which, in this case is matches the execution count so that PREGEXEC finds that all error reports are current.

The following line gives the configuration of the system, and shows that no processors are failed (i.e. retired). The following 6 output items are the binary representations of the six processor error reports. The error report from processor 0 is marking itself for retirement; the report from processor 1 agrees. No processors are indicated as faulty in the error reports of processors 2 and 3, but processors 4 and 5 indicate that processor 1 should be retired.

The next item is the reconfiguration word generated by the primary global executive. It indicates that processor 0 should be retired, and that the current frame count is 1 (in bit position 8). The global executive acceptance test detects an error, and invokes the alternate routine, which marks processor 1 for retirement as shown in the last output item.

This particular case demonstrates that the acceptance test can detect the error of simultaneously incorrectly marking a functional processor as for retirement (processor 0) and not detecting a failed processor (processor 1). The second case shown in figure C.2 shows that processors 0, 1, 4, and 5 all indicate that processor 1 should be retired, but that the primary reconfiguration word marks processor 0 for retirement. Once again, the recovery block can detect and correct the error.

Close to 2,000 cases of this type were run, and in order to reduce the amount of output, GEXEC was modified to show only the case title, whether or not a processor which should have been retired was still generating error reports, whether the primary global executive output was accepted, and if not, the value of the alternate acceptance test was shown. Figure C.3 shows the beginning of such an output for failure to detect one faulty processor when one other was retired. The first item on the page is the prompt generated by the modified GEXEC program for the data file name. The next items show that the reconfiguration word is given as 0 throughout the file (i.e. no processors are marked for retirement by the primary global executive in this set of test cases) and that processor 1 is indicated as not working. The set of possibilities generated within GEXEC did not exclude processors marking themselves for retirement or having not working processors generating error reports. Thus, the first test case of figure C.3, processors 0 and 1 marking processor 0 for retirement. Because this condition would not lead to the retirement of processor 0, the primary error word is correct. In the second case, processor 1 is indicated as having excess disagreements by processors 0 and 1. Because processor 1 should have been retired, this is possibly a serious condition, and the global executive indicates that there may be a problem (by itself, the global executive can not diagnose and trace the problem) to the system in the message "retired processor working". In the third case, the error report from a retired processor along with only one other processor indicates that a third should be marked for retirement. This is not a sufficiently strong case for retiring processor 2, so the reconfiguration word is correct.

```
proc 0 outvoted; procs 0 1 reporting
Case  1  Enter framecount 1

Failed processors
  0   0   0   0   0   0
error report for processor  0
0000 0001 0000 0001

error report for processor  1
0000 0001 0000 0001

error report for processor  2
0000 0001 0000 0000

error report for processor  3
0000 0001 0000 0000

error report for processor  4
0000 0001 0000 0010

error report for processor  5
0000 0001 0000 0010

Reconfiguration Word
0000 0001 0000 0001

alternate reconf word
0000 0000 0000 0010
```

Figure C.2. Global Executive Validation Output

```
proc 1 outvoted; procs 0 1 reporting
Case  2  Enter framecount 2

Failed processors
  0   0   0   0   0   0
error report for processor  0
0000 0010 0000 0010

error report for processor  1
0000 0010 0000 0010

error report for processor  2
0000 0010 0000 0000

error report for processor  3
0000 0010 0000 0000

error report for processor  4
0000 0010 0000 0010

error report for processor  5
0000 0010 0000 0010

Reconfiguration Word
0000 0011 0000 0001

alternate reconf word
0000 0000 0000 0010
```

Figure C.2. (continued) Global Executive Validation Output

```
reconf and working held constant
Reconfiguration word (reconf)
0000 0001 0000 0000

Processor statuses; 0 working/ 1 failed
  0  1  0  0  0  0



proc 0 outvoted; procs 0 1 reporting
global Executive OK

proc 1 outvoted; procs 0 1 reporting
retired proc.  working
global Executive OK

proc 2 outvoted; procs 0 1 reporting
global Executive OK

proc 3 outvoted; procs 0 1 reporting
global Executive OK

proc 4 outvoted; procs 0 1 reporting
global Executive OK

proc 5 outvoted; procs 0 1 reporting
global Executive OK

proc 0 outvoted; procs 0 2 reporting
global Executive OK

proc 1 outvoted; procs 0 2 reporting
retired proc.  working
retired proc.  working
global Executive OK

proc 2 outvoted; procs 0 2 reporting
global Executive OK

proc 3 outvoted; procs 0 2 reporting
alternate reconf word
0000 0000 0000 1000

proc 4 outvoted; procs 0 2 reporting
alternate reconf word
0000 0000 0001 0000

proc 5 outvoted; procs 0 2 reporting
alternate reconf word
0000 0000 0010 0000

proc 0 outvoted; procs 0 3 reporting
global Executive OK

proc 1 outvoted; procs 0 3 reporting
retired proc.  working
retired proc.  working
global Executive OK

proc 2 outvoted; procs 0 3 reporting
alternate reconf word
0000 0000 0000 0100
```

Figure C.3. Global Executive (VALGEX) Validation Output

REFERENCES

AERO78

Aerospace Corp., Fault Tolerant Software Study, NASA Contractor Report No. 145298, Advanced Programs Division, Aerospace Corp., February, 1978

HECH76

H. Hecht, "Fault Tolerant Software for Real Time Applications", ACM Computing Surveys, Vol. 8, No. 4, p. 391, December, 1976

RAND75

B. Randell, "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, p. 220, June, 1975

1. Report No. 165874	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Fault Tolerant Software Modules for SIFT		5. Report Date April 1981
		6. Performing Organization Code
7. Author(s) Myron Hecht and Herbert Hecht		8. Performing Organization Report No. SoHaR-TR-81-04
9. Performing Organization Name and Address SoHaR, Inc. 1040 South LaJolla Ave. Los Angeles, CA 90035		10. Work Unit No.
		11. Contract or Grant No. NAS1-15428
12. Sponsoring Agency Name and Address NASA-Langley Research Center Hampton, VA 23665		13. Type of Report and Period Covered Final Engineering Report
		14. Sponsoring Agency Code
15. Supplementary Notes		

16. Abstract

The Recovery Block technique for fault-tolerant software was applied to the operating system of the SIFT fault-tolerant computer. The original operating system serves to implement algorithms for hardware fault tolerance, and has been subjected to rigorous logical analysis, but does not incorporate redundancy for tolerating its own faults, (e.g., programming errors). Fault-tree analysis was used to validate acceptance tests for application to alternate (redundant) versions of several operating system functions. The tests and several alternate program versions were implemented in Pascal. This application of the Recovery Block technique was more difficult than usual because the subject program was essentially logical in nature. Some limitations were encountered in constructing alternate routines that are truly independent of the primary ones and also of the acceptance test.

17. Key Words (Suggested by Author(s)) Fault-Tolerant Software, Recovery Blocks, Fault-Tolerant Computers, Fault-Tree Analysis, Software Validation		18. Distribution Statement	
19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price

N-305